Masterarbeit in Physik

All-Atom Event-Chain Monte Carlo

—

Designing a General-Purpose Python Application

von

Philipp Höllmer

Angefertigt im Physikalischen Institut

Vorgelegt der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität
Bonn

Oktober 2019

I hereby declare that the work presented here was formulated by myself and that no sources or tools other than those cited were used.

Bonn, ...
Date Signature

1. Gutachter: Prof. Dr. Hartmut Monien
2. Gutachter: Prof. Dr. Simon Trebst

Acknowledgements

I would like to start by thanking Prof. Dr. Hartmut Monien. He established the first contact with Prof. Krauth, made it possible to continue to work in collaboration with him, and was always patient with me during the year of my Master thesis. I can surely say that without him it would not have been possible for me to learn so much about the exciting subject of this thesis. On top of that, he always had an open door as a mentor. Not only did he taught me a lot about physics, but also about highly interesting other areas I never would have learned about otherwise (ranging from nice Haskell tricks to old Japanese movies).

I would also like to thank Prof. Dr. Werner Krauth. On the one hand, he gave me the opportunity to experience three exciting months at the École Normale Supérieure in Paris (and to get to know my favorite Vietnamese restaurant during this time). On the other hand, he agreed on continuing to work with me, even after I went back to Bonn. Of course, he also taught me everything I know about the event-chain Monte Carlo algorithm and infected me with his enthusiasm for this subject.

A big thank you also to David Berghaus and Liang Qin. They read drafts of this thesis and gave very helpful comments, which definitely improved it a lot (at least in my humble opinion). Another thank you to Liang Qin, Dr. A. C. Maggs, and Dr. Michael F. Faulkner for always being happy to answer my many questions concerning the event-chain Monte Carlo algorithm.

I acknowledge support from the Bonn-Cologne Graduate School of Physics and Astronomy honors branch and from the Institut Philippe Meyer.

Finally, thank you to my family who always supported me every step of my way. And last but not least, the greatest thank you to Joana. Thank you for giving comments and advice on this thesis. Most importantly, however, thank you for being my home.

Contents

CHAPTER 1

Introduction

Markov-chain Monte Carlo methods, like the well known Metropolis algorithm [1], and molecular dynamics [2] play a central role in simulations in the field of statistical mechanics, which aims to understand the relation between the *macroscopic* properties of systems and the *microscopic* properties of its constituents. The studied systems may display fascinating collective phenomena and phase transitions (examples range from ferromagnetism [3] to superconductivity [4]). A basic assumption of statistical mechanics is that, in equilibrium, the system visits each of its microscopic states in a stochastic way during its evolution. Therefore, complex systems can be described by enumerating all possible microscopic states weighted by the probability to find the system in a particular one [5]. This corresponds analytically to the often impossible task of computing sums over the complete configuration space (or solving high-dimensional integrals).

Markov-chain Monte Carlo methods allow to reproduce the statistical behavior of a system in thermodynamic equilibrium numerically. It samples configurations of the system according to the appropriate probability distribution. These samples are then used to compute thermodynamic properties of the system. Within the computational context, equilibrium usually means that all probability flows vanish. This is enforced by the detailed-balance condition of Markov chains [6]. However, this condition leads to time-reversible (diffusive) dynamics which often produce Markov chains converging only slowly to equilibrium. Similarly, the generation of independent samples, once equilibrium is reached, requires long chains [7].

In order to not be influenced by these effects, the event-chain Monte Carlo (ECMC) algorithm [8, 9] constructs an irreversible continuous-time Markov chain. It violates detailed balance and only satisfies the weaker global-balance condition. Although the configurations generated at large times sample the desired equilibrium distribution, the asymptotic state has non-vanishing probability flows. In an interacting particle system with periodic boundary conditions, for example, the particles might move preferentially in certain directions even though the configurations visited in the Markov chain realize the equilibrium distribution. Moreover, there are no rejected configurations during the evolution of the Markov chain.

ECMC simulations often equilibrate faster than their reversible counterparts [10–14]. They solved the puzzle about the precise nature of the melting phase transition of two-dimensional hard disks [15, 16], after decades of unsuccessful studies using other Monte Carlo approaches as well as molecular dynamics simulations. The ECMC algorithm is conceived for systems of constituents described by continuous variables. The constituents might take part in (soft and hard) distance-dependent

many-body interactions [9, 17] and can furthermore be placed in an external potential [18, 19]. Besides hard disks, it was successfully applied to soft disks [20], spin models [10, 12, 21], problems in quantum-field theory [22], and dense all-atom systems with long-range Coulomb interactions [7].

ECMC combines three concepts: First, the factorized Metropolis filter [9, 17], which replaces the traditional Metropolis acceptance criterion by a consensus rule. It applies to models where the probability distribution can be split into a product over sets of independent and separately treated factors. Each of these factors only depends on a subset of all constituents of the system and a proposed configuration must be accepted by all of them.

Second, the lifting framework [23, 24] where the physical configuration space is augmented with auxiliary variables that resemble the velocity of Newtonian mechanics. Each active constituent with a non-zero velocity moves infinitesimally and persistently until a move is vetoed by a single factor, i.e., the proposed configuration generated by the moves of the active constituents is rejected by this factor. The velocity is then transferred to another constituent taking part in this factor. These lifting moves replace the characteristic rejections of the original Metropolis algorithm.

Third, the event-driven implementation, which is reminiscent of event-driven molecular dynamics [2, 25, 26]. Instead of moving the active constituents in small steps and checking if all factors accept the changed configuration, each factor determines an event time at which consensus is broken and a lifting move takes place. The earliest event time then determines the updated configuration and the relevant lifting move. With these ideas, ECMC implements the time evolution of a piecewise non-interacting (and therefore deterministic) system [27, 28].

The described concepts have remarkable implications. ECMC does not evaluate the equilibrium probability distribution or its ratios. Hence, for the Boltzmann distribution it never computes the total potential or its changes. Instead, the simulation proceeds from one event to the next with $\mathcal{O}(1)$ complexity using the cell-veto algorithm [7, 29], even for long-range potentials like the Coulomb interaction.

Moreover, ECMC samples the Boltzmann (canonical) distribution in a non-physical time evolution which is used to compute thermodynamic averages. In contrast, the time evolution of molecular dynamics (given by Newton's law) is energy-conserving (micro-canonical) and has a physical meaning. (In order to yield the Boltzmann distribution, systems in molecular dynamics simulations are generally coupled to a thermostat [30].) This allows to compute dynamic properties like time correlations or coarsening but gives no additional freedom to accelerate the exploration of the configuration space from an algorithmic point of view. Monte Carlo methods, in general, are only constrained by the global-balance condition and well chosen Monte Carlo dynamics can speed up the sampling of the equilibrium distribution: a freedom exploited by ECMC.

Another important implication is that ECMC neither discretizes space nor time. Event-driven molecular dynamics, in contrast, rely on piecewise constant potentials. Interaction potentials, for example in biophysical simulations, have therefore been coarsely discretized in order to fit into this framework [31, 32]. Although time-driven molecular dynamics do not have this restriction, the simulation has to proceed in discretized time steps instead [30].

In ECMC, most constituents are at rest (their lifting velocity is zero) but generally an arbitrary fixed number of independent active constituents with identical velocity vectors can be chosen. Most previous applications of ECMC, however, chose a single active constituent. In event-driven molecular dynamics all constituents typically have non-zero velocities. It is remarkable that it was shown nevertheless that ECMC with a single active constituent mixes and relaxes comparable to molecular dynamics for the case of one-dimensional particles with local interactions [14].

Most ECMC applications so far chose only a single active constituent because a *parallel* ECMC algorithm, which treats several active constituents independently in multiple threads or processes, is yet to be developed. So far, it was only implemented for short-ranged interactions by using a domain decomposition into stripes [33] or cells [34]. In contrast, parallel event-driven molecular dynamics algorithms have already been developed [35–37].

This thesis concerns the design of a general-purpose Python application, named JeLLyFysh (JF), that implements ECMC for a wide range of all-atom systems. Here, the full quantum-mechanical many-body system is projected onto the reduced classical degrees of freedom of the involved particles' positions. The potential energy of a configuration is thus a function of these positions. Hence, JF is conceived for systems ranging from particles interacting by central potentials to composite point objects such as finite-size dipoles, water molecules, and eventually peptides and polymers.[1] By this, it might prove useful in domains that have traditionally been reserved to molecular dynamics (in particular, the all-atom Coulomb problem in biophysics and electrochemistry).

The application's architecture closely mirrors the mathematical formulation presented in [7, Section II]. The first version JeLLyFysh-Version1.0 (JF-V1.0) includes homogeneous translation-invariant N-body systems in a regularly shaped simulation box with periodic boundary conditions. The interactions can possibly be long-ranged and optionally be treated by the cell-veto algorithm. The architecture of the application is designed to remain easily expendable toward new interaction potentials. It is furthermore suited to treat, for example, spin models and problems in quantum-field theory. The application is intended to grow into a basis code that will foster the development of irreversible Markov-chain algorithms, which have shown their strong potential in the form of the ECMC algorithm.

I had the great pleasure to design the application in collaboration with Liang Qin (ENS Paris), Michael F. Faulkner (University of Bristol), A. C. Maggs (ESPCI Paris) and Werner Krauth (ENS Paris). The results have been published [39]. (The publication is also attached, see publication in Appendix B.) The application is developed as an open-source project on GitHub and can be executed by any Python implementation which supports Python version 3.5 or higher.[2] It is made available under the GNU GPLv3 license. Contributions to the application via pull requests are encouraged.

The remainder of this thesis is structured as follows: Chapter 2 gives a mathematical introduction into ECMC. Chapter 3 describes the architecture of the application, which is based on the mediator design pattern [40]. The following Chapter 4 discusses how the eponymous events of ECMC are determined in the event handlers of JF. Afterwards, Chapter 5 presents the user interface realized through configuration files and how system definitions (such as, for example, the simulation box) are broadcast to the different elements of the application. In Chapter 6 a number of worked-out simulation examples for systems of atoms, dipoles, and water molecules are discussed, before the thesis is finally concluded in Chapter 7.

[1] The name implies that the application treats simulations of 95.56 % water and 4.44 % other things, which is the chemical composition of the *Aurelia aurita* jellyfish [38].

[2] The URL of the repository is https://github.com/jellyfysh/JeLLyFysh.

CHAPTER 2

Event-Chain Monte Carlo Algorithm

This chapter introduces the mathematical formulation of ECMC (see [7, Section II]). It begins with a short introduction to Monte Carlo methods (see Section 2.1). The difference between direct sampling and Markov-chain sampling is emphasized and an algorithm from each category is presented (rejection sampling and the Metropolis algorithm). The application of these algorithms to the hard-sphere model in Section 2.2 showcases some of their limitations and motivates the development of ECMC. The factorized Metropolis filter, the lifting framework, and the event-driven implementation are thoroughly explained in the Sections 2.3–2.5, followed by a presentation of the cell-veto algorithm in Section 2.6. Finally, Section 2.7 applies these concepts again to the hard-sphere model.

2.1 Introduction to Monte Carlo Methods

Monte Carlo methods can be used as a statistical approach to compute expected values of a quantity $\mathcal{A}$,

$$\langle \mathcal{A} \rangle = \sum_{c \in \Omega} \pi(c)\, \mathcal{A}(c). \tag{2.1}$$

Here, Ω is the discrete space of all possible configurations c, $\mathcal{A}(c)$ is the value of the quantity $\mathcal{A}$ in the configuration c, and π is the normalized probability distribution of the configurations. Normalized means that

$$\sum_{c \in \Omega} \pi(c) = 1. \tag{2.2}$$

Monte Carlo methods are used when the configuration space is too large to directly carry out the summation in Eq. (2.1). (For the case of a continuous space of configurations Ω, the sums in Eqs. (2.1) and (2.2) are replaced by an integral over the complete space.)

In statistical mechanics, the probability of a configuration c is often given by the Boltzmann distribution

$$\pi_B(c) = \frac{1}{Z} \mathrm{e}^{-\beta U(c)}, \tag{2.3}$$

where $U(c)$ is the total potential of the configuration and β is the thermodynamic beta. Z is the canonical partition function and $\langle \mathcal{A} \rangle$ is then usually referred to as the canonical ensemble average. The probability of a configuration is usually not exactly known because of the denominator.

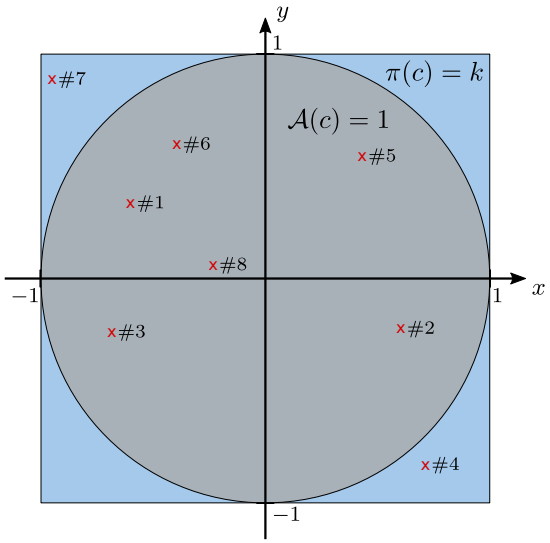


Figure 2.1: A direct sampling simulation to determine the numerical value of the number π. The probability distribution is given by $\pi(c) = k > 0$ for configurations $c = (x, y) \in \mathbb{R}^2$ located in the blue square and $\pi(c) = 0$ otherwise. Similarly, $\mathcal{A}(c) = 1$ only inside of the gray circle. The red crosses show eight random independent samples from which two are located outside the circle (samples #4 and #7). Equation (2.4) yields an approximation for the ratio of the areas of the circle and the square and therefore $\pi \approx 3$.

2.1.1 Direct Sampling

The approximation of the expected value $\langle \mathcal{A} \rangle$ via Monte Carlo methods is based on the generation of N_S samples of configurations according to their probability [41]. Obtaining the samples *independently* from each other is commonly referred to as direct sampling. Suppose the generated configurations are $c_1, \ldots, c_N \in \Omega$. The (finite) expectation value $\langle \mathcal{A} \rangle$ can then be approximated by the sample average $\overline{\mathcal{A}}_{N_S}$,

$$\langle \mathcal{A} \rangle \approx \overline{\mathcal{A}}_{N_S} := \frac{1}{N_S} \sum_{i=1}^{N_S} \mathcal{A}(c_i). \tag{2.4}$$

That is because according to the strong law of large numbers the sample average converges almost surely to the expected value [42]. In mathematical terms,

$$\text{Probability}\left(\lim_{N_S \to \infty} \overline{\mathcal{A}}_{N_S} = \langle \mathcal{A} \rangle \right) = 1. \tag{2.5}$$

If the variance $\text{Var}(\mathcal{A}) = \langle \mathcal{A}^2 \rangle - \langle \mathcal{A} \rangle^2$ is finite, the central limit theorem furthermore ensures that the distribution of sample averages in the limit $N_S \to \infty$ approaches a Gaussian distribution with mean $\langle \mathcal{A} \rangle$ and variance $\text{Var}(\mathcal{A})/N_S$ [6].

A well known example uses direct sampling to determine the numerical value of the number π. Consider the two-dimensional continuous space of configurations $c = (x, y) \in \mathbb{R}^2$. The geometric center of a square with side length 2 is located at the position $(0, 0)$. At the same position, the geometric center of a circle with diameter 2 is located. Let $\pi(c) = k > 0$ for all configurations within the square and $\mathcal{A}(c) = 1$ for all configurations within the circle. Taking two uniformly distributed real random

numbers between -1 and 1 generates independent random samples of configurations according to $\pi(c)$. In each of these samples, the configuration can be within the circle or not. For a large number of samples the ratio of the number of samples within the circle and the total number of samples yields an approximation for the ratio of the areas of the circle and the square. This can be used to approximate the number π (see Fig. 2.1).

A general way to construct independent samples according to a probability distribution π is the inverse transform sampling [43, 44]. First, assume that the possible configurations are described by a single real number, $c = x \in \mathbb{R}$.[1] The cumulative distribution function $F(x)$ is then defined as the probability that c takes values less than or equal to x,

$$F(x) \coloneqq \int_{-\infty}^{x} \mathrm{d}x' \, \pi(x'). \tag{2.6}$$

F is at least weakly monotonic and its inverse is defined for arguments $0 < v < 1$ as

$$F^{-1}(v) \coloneqq \inf \left\{ x' : F(x') \geq v \right\}. \tag{2.7}$$

If v is chosen as a uniformly distributed random number between 0 and 1, $F^{-1}(v)$ is distributed according to the probability distribution $\pi(x)$. This can be extended in a straightforward manner to the discrete configuration space, but the extension to multivariate distributions (that is, probability distributions which depend on more than a single parameter) is quite cumbersome [44]. Assume that there are d parameters. The analog of the inversion transfer sampling method (the conditional distribution method) is only possible if the probability distribution can be written as

$$\pi(x_1, \ldots, x_d) = \pi_1(x_1)\pi_2(x_2|x_1) \ldots \pi_d(x_d|x_1, \ldots, x_{d-1}). \tag{2.8}$$

Here, $\pi_d(x_d|x_1, \ldots, x_{d-1})$ denotes the conditional probability for x_d given the other values $x_1, \ldots, x_{d-1}$. Furthermore, all of the conditional probability distributions must be invertible so that the inverse transform sampling can be applied to each of them. For the Boltzmann distribution this method is not suitable because, as mentioned, the canonical partition function and therefore the exact probability of a configuration is generally not known. (Also, if the canonical partition function was known, there would be hardly any need for a Monte Carlo simulation anyways.) However, the inverse transform sampling is often used to generate non-uniformly distributed random numbers, for example following an exponential distribution.

In order to directly sample the Boltzmann distribution, rejection sampling can be used [43, 44]. Consider, for example, classical particles in a box. The interactions between the particles and the external potential yield the total potential $U(c) \geq 0$ of a configuration. To generate samples according to $\pi_B(c)$, first a configuration from a uniform distribution is sampled, i.e., each particle is placed at a random position in the box. Then, this sample is accepted with the probability $\exp(-\beta U(c))$ and only the accepted samples are used to compute the canonical ensemble average. It is important to note that the acceptance probability does not depend on the canonical partition function. The exact value of the probability of a configuration is not relevant but only the relative probabilities. The rejection sampling algorithm is not practical in most cases because the acceptance probabilities of the generated configurations $\exp(-\beta U(c))$ are often very small [1].

[1] In this context, one usually calls the possible configurations the possible values of a continuous random variable. For the sake of simplicity, the already introduced terms are kept.

2.1.2 Markov-Chain Sampling

The well known Metropolis algorithm [1] creates samples of configurations according to a probability distribution using a Markov chain [45]. A brief discussion of a Markov chain is given in the following (see [46, Section 2.2.4], for a more formal discussion see [47]).

Consider a finite set of possible configurations $c_1, c_2, c_3, \ldots$ (the discussion can be generalized to a continuum of configurations). Define a stochastic process at discrete times labeled consecutively $t_1, t_2, t_3, \ldots$, and denote by X_t the configuration of the system at time t. Similar to Eq. (2.8), the conditional probability that $X_{t_n} = c_{i_n}$ is written as

$$\pi(X_{t_n} = c_{i_n} | X_{t_{n-1}} = c_{i_{n-1}}, X_{t_{n-2}} = c_{i_{n-2}}, \ldots, X_{t_1} = c_{i_1}), \tag{2.9}$$

given that at the preceding time $X_{t_{n-1}} = c_{i_{n-1}}$, etc. If the conditional probability is independent of all configurations except the immediate predecessor, that is,

$$\pi(X_{t_n} = c_{i_n} | X_{t_{n-1}} = c_{i_{n-1}}, X_{t_{n-2}} = c_{i_{n-2}}, \ldots, X_{t_1} = c_{i_1}) = \pi(X_{t_n} = c_{i_n} | X_{t_{n-1}} = c_{i_{n-1}}), \tag{2.10}$$

the stochastic process is called a Markov process and the sequence of configurations is called a Markov chain. The given conditional probability can then be interpreted as a transition probability $p(i \to j)$ to move from the configuration c_i to the configuration c_j,

$$p(i \to j) = \pi(X_{t_n} = c_j | X_{t_{n-1}} = c_i). \tag{2.11}$$

The transition probabilities should obey

$$p(i \to j) \geq 0, \quad \text{and} \quad \sum_j p(i \to j) = 1. \tag{2.12}$$

The total probability that at the time t_n the system is in the configuration c_j is

$$\pi(X_{t_n} = c_j) = \sum_i p(i \to j)\, \pi(X_{t_{n-1}} = c_i). \tag{2.13}$$

The Metropolis algorithm generates a new configuration from a previous configuration and therefore constructs a Markov chain. In order to converge toward the wanted probability distribution, the transition probabilities must be chosen so that Eq. (2.13) is satisfied. Introducing the flow $\mathcal{F}$ and omitting the time notation, this equation is written as

$$\mathcal{F}_c := \sum_{c'} \mathcal{F}_{c' \to c} := \sum_{c'} p(c' \to c)\, \pi(c') = \pi(c), \tag{2.14}$$

which states that the total flow into a configuration c, given by the sum of the flows from all other configurations c' into c, must equal its probability. This condition is called the global-balance condition. A special solution of Eq. (2.14) is given by the detailed-balance condition,

$$p(c \to c')\, \pi(c) = p(c' \to c)\, \pi(c'). \tag{2.15}$$

This can be easily shown by using the normalization property of the transition probabilities in

Eq. (2.12). Detailed balance expresses that the flow from a configuration c' into another configuration c equals the inverse flow. A *reversible* Markov chain obeys the detailed-balance condition.

In addition to the global-balance condition, the Markov chain must also be ergodic. This ensures that the time average over the configurations visited during the Markov chain equals the configuration space average. A chain is ergodic if it is possible to eventually reach any configuration from any other configuration with non-zero probability [48]. (For a detailed discussion when Markov chains fail to be ergodic, see the same reference.) Although non-ergodic simulations are sometimes desired (for example below a phase transition), one should always take care that the implemented simulation is not *unintentionally* non-ergodic [46]. In order for a Markov chain to converge toward the desired stationary equilibrium, it must also be aperiodic. A configuration has the period τ if any return to the configuration occurs in multiples of τ time steps. The Markov chain is aperiodic if all configurations have the period $\tau = 1$ [48]. In this thesis, it is generally assumed that the generated Markov chains are aperiodic and it is just argued that the presented algorithms yield in principle ergodic chains.

2.1.2.1 Metropolis Algorithm

The Metropolis algorithm uses a symmetric proposal distribution $g^{\mathrm{Met}}(c \to c') = g^{\mathrm{Met}}(c' \to c)$ to propose a new configuration. Afterwards, the new configuration is accepted with the probability

$$A^{\mathrm{Met}}(c \to c') = \min\left[1, \frac{\pi(c')}{\pi(c)}\right]. \tag{2.16}$$

The transition probabilities $p^{\mathrm{Met}}(c \to c') = g^{\mathrm{Met}}(c \to c')A^{\mathrm{Met}}(c \to c')$ satisfy the detailed-balance condition in Eq. (2.15). This can be shown straightforwardly by considering the distinct cases $\pi(c) > \pi(c')$, $\pi(c) < \pi(c')$ and $\pi(c) = \pi(c')$. (Non-symmetric proposal distributions are considered in the Metropolis-Hastings algorithm [49].)

For the Boltzmann distribution in Eq. (2.3), the Metropolis algorithm accepts the new configuration c' with the probability $\min\left[1, \exp(-\beta\Delta U)\right]$, where $\Delta U = U(c') - U(c)$. The acceptance rate can be changed by adjusting the symmetric proposal distribution (see Section 2.2). In the original paper, Metropolis considered N particles, each of them interacting via a pair potential. A new configuration is proposed based on each particle in succession by embedding it into a square with the side length 2δ and choosing a new position within this square. Since a particle is allowed to move to any point in the square with a finite probability, a large enough number of moves will enable it to reach any point in the complete simulation box. This is true for all particles and any configuration in the phase space can eventually be reached. Hence, the method is ergodic for such systems [1].

There had been great progress in Markov-chain Monte Carlo methods since the development of the Metropolis algorithm. Non-local cluster algorithms were used, among other systems, in classical spin models [50, 51] and systems where the total potential is either infinite or zero (which is the case, for example, for the hard-sphere model in Section 2.2) [52]. Extensions of the classical detailed-balance condition led to faster converging Markov chains by introducing persistence between subsequent moves. This solves the problem that detailed balance yields diffusive time-reversible Markov-chain dynamics. Examples include the guided walk Metropolis algorithm [53], hybrid Monte Carlo [54], and the overrelaxed Markov-chain Monte Carlo algorithm [55]. The lifting framework [23, 24] unifies these concepts. Here, the physical configuration space Ω is augmented with auxiliary variables that resemble the velocity of Newtonian mechanics. This framework is used in ECMC (see Section 2.4).

2.1.3 Data Analysis

There is an essential difference between the direct sampling presented in Section 2.1.1 and the Markov-chain sampling discussed in Section 2.1.2. Direct sampling yields independent uncorrelated samples of the desired probability distribution whereas Markov-chain sampling yields correlated samples. This section begins with the error analysis of direct sampling which is straightforward due to the central limit theorem [46]. The sample average in Eq. (2.4) is an unbiased estimator of the expectation value $\langle \mathcal{A} \rangle$. In the following, an unbiased estimator for the variance of the distribution of the sample averages,

$$\mathrm{Var}(\overline{\mathcal{A}}_{N_S}) = \langle \overline{\mathcal{A}}_{N_S}^2 \rangle - \langle \overline{\mathcal{A}}_{N_S} \rangle^2, \tag{2.17}$$

is constructed. Here, the expectation value is obtained by averaging over the ensemble of all possible sets of samples of size N_S. Inserting Eq. (2.4) into Eq. (2.17) yields

$$\mathrm{Var}(\overline{\mathcal{A}}_{N_S}) = \frac{1}{N_S^2} \sum_{i=1}^{N_S} \sum_{j=1}^{N_S} \gamma_{i,j} = \frac{1}{N_S^2} \left(\sum_{i=1}^{N_S} \gamma_{i,i} + \sum_{i=1}^{N_S} \sum_{j=1, j\neq i}^{N_S} \gamma_{i,j} \right), \tag{2.18}$$

with the unnormalized autocorrelation function of the observable $\mathcal{A}$

$$\gamma_{i,j} \coloneqq \langle \mathcal{A}_i \mathcal{A}_j \rangle - \langle \mathcal{A}_i \rangle \langle \mathcal{A}_j \rangle. \tag{2.19}$$

For uncorrelated samples, $\gamma_{i,j} = 0$ if $i \neq j$. Together with $\langle \mathcal{A}_i \rangle = \langle \mathcal{A} \rangle$ and $\langle \mathcal{A}_i^2 \rangle - \langle \mathcal{A}_i \rangle^2 = \mathrm{Var}(\mathcal{A})$, the variance of the distribution of sample averages becomes

$$\mathrm{Var}(\overline{\mathcal{A}}_{N_S}) = \frac{\mathrm{Var}(\mathcal{A})}{N_S}. \tag{2.20}$$

This result agrees with the central limit theorem but the latter is more specific as it also fixes the shape of the distribution of the sample averages. A *biased* estimator of $\mathrm{Var}(\mathcal{A})$ is given by the sample variance

$$\overline{\mathrm{Var}(\mathcal{A})}_{N_S} \coloneqq \frac{1}{N_S} \sum_{i=1}^{N_S} \left(\mathcal{A}_i - \overline{\mathcal{A}}_{N_S} \right)^2. \tag{2.21}$$

To prove that this estimator is biased, consider the expectation value which can be written as

$$\begin{aligned}
\langle \overline{\mathrm{Var}(\mathcal{A})}_{N_S} \rangle &= \left\langle \frac{1}{N_S} \sum_{i=1}^{N_S} \left(\mathcal{A}_i - \overline{\mathcal{A}}_{N_S} \right)^2 \right\rangle \\
&= \frac{1}{N_S} \sum_{i=1}^{N_S} \left\langle \mathcal{A}_i^2 - \frac{2}{N_S} \mathcal{A}_i \sum_{j=1}^{N_S} \mathcal{A}_j + \frac{1}{N_S^2} \sum_{j=1}^{N_S} \mathcal{A}_j \sum_{k=1}^{N_S} \mathcal{A}_k \right\rangle \\
&= \frac{1}{N_S} \sum_{i=1}^{N_S} \left[\frac{N_S - 2}{N_S} \langle \mathcal{A}_i^2 \rangle - \frac{2}{N_S} \sum_{j=1, j\neq i}^{N_S} \langle \mathcal{A}_i \mathcal{A}_j \rangle \right. \\
&\qquad \left. + \frac{1}{N_S^2} \sum_{j=1}^{N_S} \langle \mathcal{A}_j^2 \rangle + \frac{1}{N_S^2} \sum_{j=1}^{N_S} \sum_{k=1, k\neq j}^{N_S} \langle \mathcal{A}_j \mathcal{A}_k \rangle \right].
\end{aligned} \tag{2.22}$$

For uncorrelated samples, this can be further simplified to

$$\begin{aligned}\langle\overline{\mathrm{Var}(\mathcal{A})}_{N_S}\rangle &= \frac{1}{N_S}\sum_{i=1}^{N_S}\left[\frac{N_S-2}{N_S}\Big(\mathrm{Var}(\mathcal{A})+\langle\mathcal{A}\rangle^2\Big)-\frac{2(N_S-1)}{N_S}\langle\mathcal{A}\rangle^2\right.\\ &\qquad\left.+\frac{1}{N_S}\Big(\mathrm{Var}(\mathcal{A})+\langle\mathcal{A}\rangle^2\Big)+\frac{N_S(N_S-1)}{N_S^2}\langle\mathcal{A}\rangle^2\right]\\ &= \frac{N_S-1}{N_S}\mathrm{Var}(\mathcal{A}).\end{aligned} \tag{2.23}$$

Hence, to get an *unbiased* estimator of $\mathrm{Var}(\mathcal{A})$ the biased estimator in Eq. (2.21) must be multiplied by $N_S/(N_S-1)$. According to the central limit theorem, the sample averages are distributed like a Gaussian distribution with mean $\langle\mathcal{A}\rangle$ and variance $\mathrm{Var}(\mathcal{A})/N_S$. Thus, the sample variance can be used to estimate the 68 % confidence interval of the sample average,

$$\langle A\rangle = \frac{1}{N_S}\sum_{i=1}^{N_S}\mathcal{A}_i \pm \sqrt{\frac{1}{N_S(N_S-1)}\sum_{i=1}^{N_S}\left(\mathcal{A}_i-\frac{1}{N_S}\sum_{j=1}^{N_S}\mathcal{A}_j\right)^2}. \tag{2.24}$$

Markov-chain sampling, on the other hand, starts at an initial configuration and evolves it. This implies two problems: First, based on the initial configuration, the Markov chain needs some time in order to thermalize toward the desired probability distribution. Second, after the Markov chain has thermalized, the sample average uses correlated samples which renders Eq. (2.24) invalid.

The mixing time τ_{mix} is the time needed by the Markov chain to reach the equilibrium distribution from the worst initial configuration. Unfortunately, the exact determination of τ_{mix} is not at all an easy problem [48]. (It is possible to construct lower and upper bounds for τ_{mix} using the later defined autocorrelation time of the slowest mode, which contains information about the correlation of samples when equilibrium is already achieved. However, this upper bound is usually very large.) A practical estimate of τ_{mix} can be obtained by the following procedure: Start from two opposite initial configurations (e.g., around a phase transition from an ordered and an unordered initial configuration). If the two chains converge to the same configuration after some time, it is most probably thermalized and an approximate value of τ_{mix} is found.

This strategy is formalized by the coupling of Markov chains [19, 48]. Here, a chain is started from every possible initial configuration. If two chains yield the same configuration after a time step, they are merged. After all chains were merged, all correlations with the initial configurations are lost and an upper bound for τ_{mix} is found. Clearly, this method is only applicable to specific systems. Also, in some situations, the upper bound determined by the coupling method can be far too large [56].

Once the Markov chain is thermalized, sample values of an observable can be taken. This sampling is done in fixed time intervals δt (for the Metropolis algorithm this corresponds to sampling after a fixed number of proposed moves). The fact that these samples are correlated breaks the central limit theorem but the strong law of large numbers still holds. Therefore the sample average still gives a good estimate of $\langle\mathcal{A}\rangle$. However, the variance of the sample average in Eq. (2.18) does not reduce to the simple form in Eq. (2.20) because the correlators $\gamma_{i,j}$ can be unequal 0, also for $i \neq j$. Nevertheless, an unbiased estimator can be found [57]. If the Markov chain is in equilibrium, the correlators are invariant under time translations and one can define $\gamma_t := \gamma_{i,j}$, where $t = |i-j|$. Counting in

Eq. (2.18) the number of pairs of i and j with the distance $d = i - j$ for values of $i, j \in \{1, \dots, N_S\}$ (with the restriction $i \neq j$) results in the equation

$$\mathrm{Var}(\overline{\mathcal{A}}_{N_S}) = \frac{1}{N_S}\left[\gamma_0 + 2\sum_{t=1}^{N_S-1}\left(1 - \frac{t}{N_S}\right)\gamma_t\right] \tag{2.25}$$

for the variance of the distribution of the sample averages. For $d = 1$, there are $N_S - 1$ pairs (fix j and let $i = j + 1$, which is always possible except for $j = N_S$). Similarly, there are $N_S - 2$ pairs for $d = 2$ and generally $N_S - d$ pairs for $d \in \{-(N_S - 1, \dots, -1, 1, \dots, N_S - 1)\}$. The absolute value in $t = |d|$ yields an additional factor of 2.

The variance of the distribution of the sample averages is often estimated by using an estimate for γ_t. However, the most obvious estimator,

$$c_t = \frac{1}{N_S - t}\sum_{i=1}^{N_S-t}(\mathcal{A}_i - \overline{\mathcal{A}}_{N_S})(\mathcal{A}_{i+t} - \overline{\mathcal{A}}_{N_S}), \tag{2.26}$$

is biased (since $\langle c_T \rangle \neq \gamma_t$). Nevertheless, these quantities can be used to construct an approximately unbiased estimator for the variance of the distribution of the sample averages [57],

$$\overline{\mathrm{Var}(\mathcal{A})}_{N_S} = \frac{c_0 + 2\sum_{t=1}^{T}(1 - \frac{T}{N_S})c_t}{N_S - 2T - 1 + \frac{T(T+1)}{N_S}}. \tag{2.27}$$

T is a cut-off parameter in the sum. To understand the underlying approximation of this estimator, the autocorrelation time τ_{cor} must be introduced. It is defined for every observable $\mathcal{A}$ as [58]

$$\tau_{\mathrm{cor}} = \lim_{t\to\infty}\sup\frac{t}{-\log|\gamma_t/\gamma_0|}. \tag{2.28}$$

This is motivated by the fact that the normalized autocorrelation function γ_t/γ_0 typically decays exponentially ($\sim \exp(-t/\tau)$) for large times t. Equation (2.28) extracts the largest time-scale of the decay of the normalized autocorrelation function. The estimator for the variance of the distribution of the sample averages in Eq. (2.27) is only a good approximation if τ_{cor} is finite so that the sum can be truncated for $T \gg \tau_{\mathrm{cor}}/\delta t$. It should also be noted that for T close to $N_S - 1$, the estimator in Eq. (2.27) is calculated by the ratio of two small numbers with a fluctuating numerator. This gives an unreasonable approximation. Thus, all together, one should choose $\tau_{\mathrm{cor}}/\delta t \ll T \ll N_S$.

In practice, the usage of Eq. (2.27) is non-trivial. The first possibility is to study the decay of c_t with increasing t, try to extract τ_{cor} and choose T based on this. In a more sophisticated way, the estimate $\overline{\mathrm{Var}(\mathcal{A})}_{N_S}$ is monitored in dependence of T up to some value $T_{\max}$. If the estimate gets constant, this value is an unbiased estimate of the variance of the distribution of the sample averages. However, c_t is essentially a random number for $t \gg \tau_{\mathrm{cor}}/\delta t$. If $T_{\max}$ was chosen very large, more such numbers would be included in the estimator which gives rise to increasing noise as T approaches $T_{\max}$. This can make the extraction of the constant value in the plot difficult. Furthermore, this approach demands the possibly time-consuming computation of c_t for all values of t between 0 and $T_{\max}$ and the choice of $T_{\max}$ cannot be automatized.

The blocking method estimates the variance of the distribution of the sample averages computationally more efficient by elegantly avoiding the computation of c_t and the choice of T_{max} [6, 57]. At first, the variance of the distribution of the sample averages is estimated assuming that the samples are *independent* [see Eq. (2.24)]. Of course, this will underestimate the true variance. Then, the set of samples is replaced with a new set by substituting two adjacent samples with their average value. The new set of samples is therefore half as large as the original one. For this set, again the apparent variance is computed assuming that the samples are independent (the average value remains unchanged). This procedure is repeated until the new set of samples only consists of two samples. Bunching the data like this makes the sets of samples increasingly independent and the apparent variance approaches the true variance of the distribution of the sample averages obtained through the Markov-chain sampling. Again, this is only true if the total observation time $N_S \delta t$ exceeds the autocorrelation time τ_{cor}.

The variance of the distribution of the sample averages and the influence of the time interval δt can be further understood by introducing the integrated autocorrelation time [58]

$$\tau_{\text{int}} = \frac{1}{2} + \sum_{t=1}^{\infty} \frac{\gamma_t}{\gamma_0}. \tag{2.29}$$

This quantity contains information about all the different time-scales that are present in the normalized autocorrelation function. With this, Eq. (2.25) can be approximated as [46]

$$\text{Var}(\overline{\mathcal{A}}_{N_S}) \approx \frac{1}{N_S} \text{Var}(\mathcal{A}) \left(1 + \frac{2\tau_{\text{int}}}{\delta t}\right). \tag{2.30}$$

(Note that constructing an estimator for τ_{int} would essentially yield the estimator for $\text{Var}(\overline{\mathcal{A}}_{N_S})$ in Eq. (2.27) [57, 59].) The variance of the distribution of the sample averages depends on the ratio $\tau_{\text{int}}/\delta t$ of the integrated autocorrelation time and the time interval between two samples. If the ratio is small, the different samples get uncorrelated and the variance approaches the one of independent samples. In contrast, if the ratio is very large, the variance grows. It is however not dependent on the time interval δt, but on the ratio $\tau_{\text{int}}/N_S \delta t$ of the integrated autocorrelation time and the total observation time. This ratio is just a measure on how many uncorrelated information can be extracted from the samples of the Markov chain. Although it would be a possibility to compute a sample after every step of the Markov chain, it does not decrease the error as two successive samples would still be highly correlated and a lot of samples would be necessary. It is noteworthy, as a final remark, that using the coupling method even fully decorrelated samples like the ones obtained by direct sampling can be created with a finite amount of backtracking (called coupling from the past) [60].

The data analysis of the correlated samples stemming from Markov chains assumed that the autocorrelation time τ_{cor} is finite. However, near a second-order phase transition the spatial correlation length diverges, associated with a divergent autocorrelation time τ_{cor} (critical slowing down) [5, 58]. Qualitatively, the information of a local update of a configuration in a single step of the Markov chain has to propagate over the complete correlation volume before a new statistically independent configuration is obtained. However, in any numerical simulation the correlation length cannot actually diverge since the system size sets an upper bound. This implies that the autocorrelation time scales with the system size (finite-size scaling [46]). Different implementations of a Markov chain can then speed up the process of obtaining independent samples in the vicinity of the phase transition (for example, reversible vs. irreversible chains or local vs. non-local cluster moves).

2.2 Hard-Sphere Model

The hard-sphere model defined below was one of the first ever systems studied using both Monte Carlo algorithms and molecular dynamics (that is, the explicit numerical integration of Newton's equations of motion) [1, 2, 61, 62]. In three dimensions, these references showed a phase transition which is, in fact, a first-order phase transition from a fluid (disordered) state to a crystalline solid (ordered) state [63]. More surprisingly, in 1962 Alder and Wainwright observed in their molecular-dynamics simulation of hard spheres in two dimensions (hard disks) a similar fluid-solid phase transition [64]. This was not expected because long-range positional (crystalline) order for two-dimensional systems with short-range interactions is forbidden [65]. In 1968, Mermin resolved the contradiction by showing that the solid in two dimensions is based on long-range *orientational* order [66]. (This means that the orientation of links between hard disks has long-range correlations.) However, the precise nature of the phase transition was still unknown until 2011. In this year, Bernard and Krauth successfully used ECMC simulations to study the melting of two-dimensional hard disks [15].

This section introduces the hard-sphere model and describes two possible simulations of this model: first, using rejection sampling and second, using the Metropolis algorithm. The latter uses a Markov chain to sample the desired probability distribution while the former samples it directly. In Section 2.7, the simulation of hard spheres using ECMC is discussed. The answer to the nature of the phase transition of hard disks will be given there only after the ECMC algorithm is understood (just like it played out historically).

Hard spheres of diameter σ in d dimensions are particles with the following pairwise interaction potential,

$$U(\mathbf{r}_1, \mathbf{r}_2) = \begin{cases} 0 & \text{if } |\mathbf{r}_1 - \mathbf{r}_2| \geq \sigma, \\ \infty & \text{if } |\mathbf{r}_1 - \mathbf{r}_2| < \sigma. \end{cases} \tag{2.31}$$

$\mathbf{r}_1, \mathbf{r}_2 \in \mathbb{R}^d$ are the d-dimensional position vectors of the center of the spheres and $|\mathbf{r}|$ denotes the euclidean norm of the vector $\mathbf{r}$. The potential yields two different kinds of configurations. In forbidden configurations at least one pair of hard spheres overlaps and the Boltzmann weight $\exp(-\beta U(c))$ that appears in the Boltzmann distribution in Eq. (2.3) is zero. Allowed configurations, in which there are no overlaps of spheres, all have the same Boltzmann weight of value one. Due to the diverging potential there is no energy scale and the phase diagram is identical at all values of β. It depends only on one parameter which is for N hard spheres in a box of volume V the packing fraction η. This is defined as the ratio of the volume occupied by the spheres and the volume of the box. In two dimensions it is given by $\eta = N\pi\sigma^2/V$.

The rejection sampling algorithm for N hard spheres to generate all allowed configurations with the same probability is constructed straightforwardly: Place each of the spheres independently in the system and reject the forbidden configurations (see Fig. 2.2). However, this algorithm gets very costly for all but the smallest and least dense systems because a great number of the generated samples are forbidden (see [6, Section 2.2.2] where only 6 out of 10^6 samples were accepted for $N = 16$ two-dimensional hard disks at the packing fraction $\eta = 0.3$).

The Metropolis algorithm for N hard spheres is implemented as follows: Start from an allowed configuration c. Then, pick a random hard sphere and sample a random displacement for each direction between $-\delta$ and δ. If the configuration c' with the displaced hard sphere is allowed, it is accepted (see Fig. 2.3). This procedure is repeated. The acceptance rate of a simulation can be changed by adjusting the parameter δ of the proposal distribution. If δ is small, the acceptance rate will be high but it will

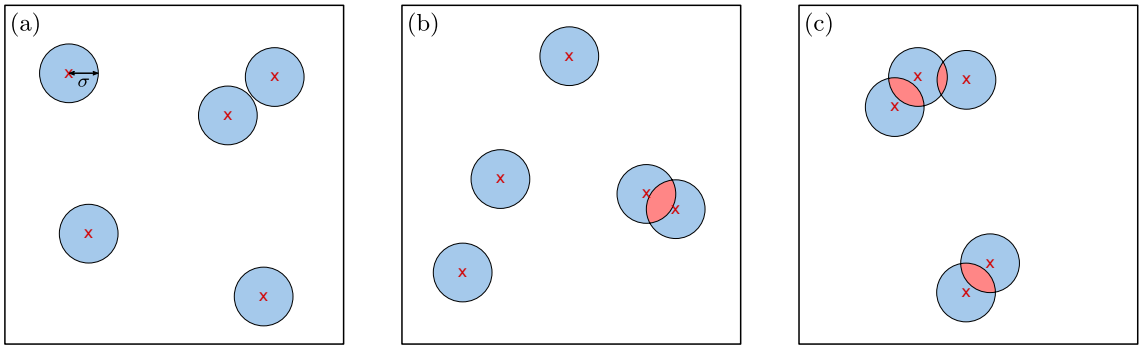


Figure 2.2: Three generated configurations of the rejection sampling algorithm for $N = 5$ hard disks at the packing fraction $\eta = 0.12$. First, the centers (shown as red crosses) of the disks with radius σ (shown as blue areas) are distributed randomly in the system. The generated configuration is accepted if there is no overlap between disks (shown as red areas). (a): An allowed configuration. (b): A forbidden configuration with one overlap between hard disks. (c): A forbidden configuration with three overlaps between hard disks.

take a long time to decorrelate since the hard spheres only travel a small path in each move. In contrast, if δ is too large a lot of moves will be rejected, which again yields on average a small traveled path in each move. (A rule of thumb is that the acceptance rate should be of the order of $1/2$ [6].) The check for overlaps after a move of a sphere can be implemented efficiently by using a cell-occupancy system. The system is divided into cells with a side length of at least 2σ. At the beginning of the simulation a map from each cell onto the spheres within the cell is established. A sphere in a given cell can then only overlap with spheres in the same (as there can be more than one sphere in the same cell) or neighbored cells. On each move, the cell-occupancy system must be kept consistent.

The Metropolis algorithm allows to sample configurations at particle numbers and densities that are far out of reach for the rejection sampling algorithm. However, upon entering the solid phase there is a considerable slowdown (see [6, Section 2.2.3] where a particular initial state with $N = 256$ hard disks at the packing fraction $\eta = 0.72$ even shows through after over 25 billion attempted moves) which makes it impossible to study the phase transition of the two-dimensional hard-disk model [15, 62].

2.3 Factorized Metropolis Filter

The ECMC algorithm, similar to the Metropolis algorithm, uses a Markov chain to sample the Boltzmann distribution. It is conceived for configurations $c = \{\mathbf{s}_1, \ldots, \mathbf{s}_i, \ldots, \mathbf{s}_N\}$ described by continuous variables $\mathbf{s}_i$, where i is a unique identifier. $\mathbf{s}_i$ may represent, for example, the position of the ith particle in the system or the angle of a spin on the ith lattice site. For concreteness, the remaining part of this chapter considers N point-like particles with positions $\{\mathbf{r}_1, \ldots, \mathbf{r}_N\}$, $\mathbf{r}_i \in \mathbb{R}^d$. The traditional Metropolis acceptance criterion of a new configuration in Eq. (2.16) is replaced by a factorized Metropolis filter which applies to models whose probability distribution can be split into sets of independent factors [9, 17]. This section uses the mathematical formulation presented in [7].

For the case of the Boltzmann distribution in Eq. (2.3), which depends on the total potential U of a configuration c, each factor M is associated to a factor potential U_M. The factors are constructed so that the total potential is written as a sum over the factor potentials. Formally, let $\mathcal{P}$ be the power set of the particle identifiers (that comprises all identifiers, pairs of identifiers, triplets, etc.) and $\mathcal{T}$ be the

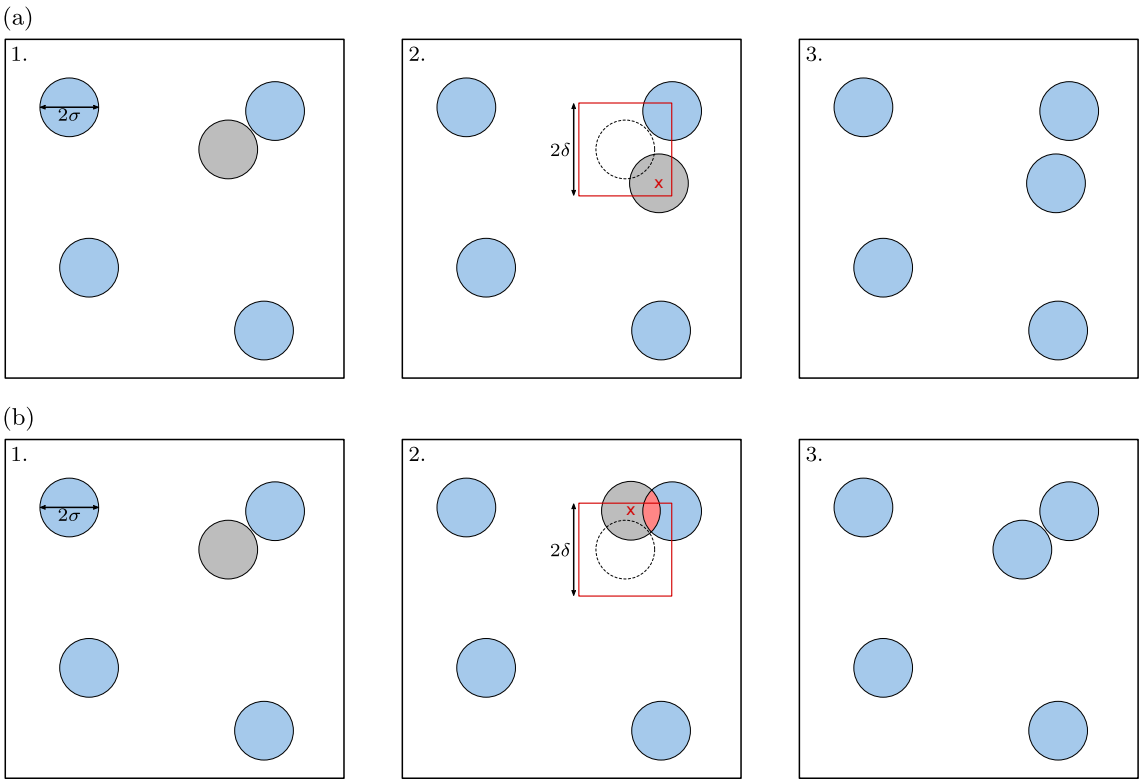


Figure 2.3: Two different moves of the Metropolis algorithm for $N = 5$ hard disks with radius σ (shown in blue) at the packing fraction $\eta = 0.12$. Each move consists of three steps: First, a random disk is picked (shown in gray). Second, a random new center position for the center of this disk (shown as a red cross) is proposed within a square with side length 2δ around the old center (shown as a red square). Third, if the new center position yields an allowed configuration with no overlaps between disks (shown as read areas), the new configuration is accepted. Otherwise it is rejected. (a): An accepted move of the Metropolis algorithm. (b): A rejected move of the Metropolis algorithm.

set of interaction types. A factor is then defined as $M := (I_M, T_M)$, where $I_M \in \mathcal{P}$ is the index set and $T_M \in \mathcal{T}$ the type of the factor. The total potential of a configuration is then given by

$$U(c = \{\mathbf{r}_1, \ldots, \mathbf{r}_N\}) = \sum_{M \in \mathcal{M}} U_M(c_M = \{\mathbf{r}_i : i \in I_M\}). \tag{2.32}$$

The set $\mathcal{M}$ only contains factors that have a non-zero contribution to the total potential for some configuration $c \in \Omega$. A factor potential U_M is of type T_M and only depends on the particle positions corresponding to the identifiers appearing in the index set I_M (the factor configuration c_M).

The deconstruction of the total potential into factor potentials is not unique. Consider, for example, the pairwise interaction of the particles via the Lennard-Jones potential [67], which approximates, e.g., the interaction between a pair of neutral atoms. The total potential of N particles is given by

$$U(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \sum_{i=1}^{N} \sum_{j=1}^{i-1} k_{\mathrm{LJ}} \left[\left(\frac{\sigma_{\mathrm{LJ}}}{|\mathbf{r}_j - \mathbf{r}_i|} \right)^{12} - \left(\frac{\sigma_{\mathrm{LJ}}}{|\mathbf{r}_j - \mathbf{r}_i|} \right)^{6} \right], \tag{2.33}$$

where k_{LJ} and σ_{LJ} are chosen constants. An obvious choice for the factors is $(\{i,j\},\mathrm{LJ})$ with the factor potential

$$U_{(\{i,j\},\mathrm{LJ})}(\mathbf{r}_i,\mathbf{r}_j) = k_{\mathrm{LJ}}\left[\left(\frac{\sigma_{\mathrm{LJ}}}{|\mathbf{r}_j-\mathbf{r}_i|}\right)^{12} - \left(\frac{\sigma_{\mathrm{LJ}}}{|\mathbf{r}_j-\mathbf{r}_i|}\right)^{6}\right] \tag{2.34}$$

for all possible pairs of particles $\{i,j\}$. It is also possible to split each of these pair factors further into $(\{i,j\},\mathrm{LJ}_{12})$ and $(\{i,j\},\mathrm{LJ}_6)$. The factor potential for the former includes only the minuend in Eq. (2.34) and the latter only the subtrahend. The choice of the factors may influence the convergence properties of ECMC, although the sampled steady state is always given by the Boltzmann distribution [7, 14]. Also, the set of factors can be infinite even for finite N [29]. As an example, consider the pairwise Coulomb interaction in a finite system with periodic boundary conditions. The interaction between the particles i and j can be regarded as a sum of separate-image Coulomb interactions between i and each periodic copy of j (from which there are infinitely many), each of these treated in a distinct factor.

Using the factorized total potential from Eq. (2.32) reduces the Boltzmann distribution in Eq. (2.3) to a product over factor weights,

$$\pi_B(c) = \frac{1}{Z}\prod_M \mathrm{e}^{-\beta U_M(c_M)}. \tag{2.35}$$

(Here and in the following, products and sums over factors M involve $M \in \mathcal{M}$.) The factorized Metropolis filter introduces a similar factorization for the acceptance probability of a move from the configuration c to c',

$$A^{\mathrm{Fact}}(c \to c') \coloneqq \prod_M \min\left[1, \mathrm{e}^{-\beta\Delta U_M(c_M\to c'_M)}\right] = \prod_M \mathrm{e}^{-\beta\Delta U_M^+(c_M\to c'_M)}, \tag{2.36}$$

with $\Delta U_M(c_M \to c'_M) \coloneqq U_M(c'_M) - U_M(c_M)$ and $x^+ \coloneqq \max(0,x)$. Strictly speaking, one should refer to $\exp(-\beta U_M)$ as a Boltzmann factor and to $\exp(-\beta\Delta U_M^+)$ as filter factor. M is just the generalized identifier for these factors. Nevertheless, M itself is called a factor for simplicity [7].

The acceptance probabilities of the factorized Metropolis filter satisfy a condition very similar to the detailed-balance condition for transition probabilities in Eq. (2.15),

$$\underbrace{\frac{1}{Z}\prod_M \mathrm{e}^{-\beta U_M(c_M)}}_{\pi_B(c)} \underbrace{\prod_F \mathrm{e}^{-\beta\left[U_F(c'_F)-U_F(c_F)\right]^+}}_{A^{\mathrm{Fact}}(c\to c')} = \underbrace{\frac{1}{Z}\prod_M \mathrm{e}^{-\beta U_M(c'_M)}}_{\pi_B(c')} \underbrace{\prod_F \mathrm{e}^{-\beta\left[U_F(c_F)-U_F(c'_F)\right]^+}}_{A^{\mathrm{Fact}}(c'\to c)}, \tag{2.37}$$

because the Boltzmann distribution and the factorized Metropolis filter factorize in the same way (i.e., the two products in Eq. (2.37) can be combined into a single product over M). A single factor then obeys this condition for the same reasoning as the transition probabilities of the Metropolis algorithm (see Section 2.1.2.1). The factorized Metropolis filter can be introduced for general probability distributions, similar to the acceptance probability of the original Metropolis algorithm. To obey the condition in Eq. (2.37), the weight must factorize like the Boltzmann distribution in Eq. (2.35).

Just replacing the acceptance probability in the Metropolis algorithm by the factorized Metropolis filter yields a less efficient algorithm. That is because the original Metropolis filter accepts a new configuration with greater or equal probabilities. However, only the factorized filter formulates a

consensus principle. To see this, view the filter as a conjunction of Boolean random variables,

$$X^{\mathrm{Fact}}(c \to c') = \bigwedge_M X_M(c_M \to c'_M), \tag{2.38}$$

where all factorwise random variables are drawn *independently*,

$$X_M(c_M \to c'_M) = \begin{cases} \text{True} & \text{if } \mathrm{ran}_M(0,1) < \mathrm{e}^{-\beta \Delta U_M(c_M \to c'_M)}, \\ \text{False} & \text{otherwise.} \end{cases} \tag{2.39}$$

Here, $\mathrm{ran}_M(0,1)$ denotes a uniformly distributed random number between 0 and 1 drawn for the factor M. The move $c \to c'$ is accepted (that is, the random variable $X^{\mathrm{Fact}}(c \to c') = \text{True}$) if all factors M accept the move independently, each of them with the probability $\exp(-\beta \Delta U^+_M(c_M \to c'_M))$.

The lifting concept will introduce a restriction to infinitesimal changes of configurations described by continuous variables (see Section 2.4) and this limit of the factorized Metropolis filter must be understood. Let the continuous position $\mathbf{r}_i$ of the configuration c be changed along the x-axis by $\mathrm{d}x_i$ which proposes a new configuration c' to be either accepted or rejected by the factorized Metropolis filter. (This can be generalized to an arbitrary infinitesimal change of the continuous variable $\mathbf{s}_i$, which includes also, for example, the continuous rotation of a spin on the ith lattice cite.) Such an infinitesimal move implies an infinitesimal change of the factor potentials for which $i \in I_M$,

$$\mathrm{d}U_M(c_M \to c'_M) = \tilde{q}_{M,i}(c_M)\,\mathrm{d}x_i, \tag{2.40}$$

with the factor derivative

$$\tilde{q}_{M,i}(c_M) \coloneqq \partial_{x_i} U_M(c_M) = \frac{\partial U_M}{\partial x_i}(c_M). \tag{2.41}$$

The factor event rate with respect to i is defined as

$$q_{M,i}(c_M) \coloneqq \beta \tilde{q}^+_{M,i}(c_M). \tag{2.42}$$

This definition can be used if one regards the acceptance probability of a single factor M, which gets

$$\mathrm{e}^{-\beta \Delta U^+_M(c_M \to c'_M)} \xrightarrow{\Delta U_M \to \mathrm{d}U_M} 1 - \beta\,\mathrm{d}U^+_M(c_M \to c'_M). \tag{2.43}$$

Therefore, the factorized Metropolis filter includes a sum over the factors,

$$A^{\mathrm{Fact}}(c \to c') = 1 - \beta \sum_M \mathrm{d}U^+_M(c_M \to c'_M). \tag{2.44}$$

If the change of configurations stems from an infinitesimal change of $\mathbf{r}_i$, this can be rewritten by using the factor event rates in Eq. (2.42) to

$$A^{\mathrm{Fact}}(c \to c') = 1 - \sum_{\{M:\, i \in I_M\}} q_{M,i}(c_M)\,\mathrm{d}x_i. \tag{2.45}$$

Hence, $q_{M,i}\,\mathrm{d}x_i$ yields the probability that the factor M breaks the consensus of Eq. (2.38) due to the change of the configuration given by the infinitesimal change $\mathrm{d}x_i$ of the position $\mathbf{r}_i$ along the x-axis.

2.4 Lifting Framework

The general idea of the lifting framework is to split each configuration into several configurations and to construct a Markov chain in the new space of configurations. Projecting back this chain onto the original space of configurations can then improve the mixing time [23, 24]. The splitting consists of augmenting the space of configurations Ω with auxiliary variables which are used to propose new configurations. In ECMC, the auxiliary variables resemble the velocity of Newtonian mechanics[2] and lend persistence to individual Monte Carlo moves. Together with the factorized Metropolis filter, this allows to construct an irreversible Markov chain on the augmented space, i.e., a chain with transition probabilities which break detailed balance and obey *maximal* global balance. The latter means that flows between two configurations are unidirectional ($\mathcal{F}_{c\to c'} = p(c \to c')\,\pi(c) > 0 \Rightarrow \mathcal{F}_{c'\to c} = 0$) and that there are no rejections ($\mathcal{F}_{c\to c} = 0$). Again, the mathematical formulation of [7] is used.

In ECMC, any physical configuration c is lifted so that it includes the lifting variables describing the active particles. For concreteness, this section assumes without loss of generality that there is a single active particle a with the normalized velocity $\hat{\mathbf{v}}_x$ in the appropriate units which moves the active particle parallel to the x-axis. The velocity stays constant throughout this section and an augmented configuration is simply written as (c, a) (although strictly speaking, the velocity should be included among the lifting variables). In principle, the Boltzmann distribution now depends on a. However, the requirement $\pi_B[(c, a)] = \pi_B(c)/N$ allows to absorb the factor $1/N$ into the normalization constant of the probability distribution. The lifted configuration (c, a) determines the proposed lifted configuration (c', a):

$$(c = \{\mathbf{r}_1, \ldots, \mathbf{r}_a, \ldots, \mathbf{r}_N\}, a) \to (c' = \{\mathbf{r}_1, \ldots, \mathbf{r}_a + \hat{\mathbf{v}}_x \Delta t, \ldots, \mathbf{r}_N\}, a). \tag{2.46}$$

In the proposed configuration (c', a) the active particle is displaced by $\hat{\mathbf{v}}_x \Delta t$ where Δt denotes a fixed time step which converts the velocity into a displacement. If (c', a) is accepted by the factorized Metropolis filter, a stays active and a new configuration with a again displaced by $\hat{\mathbf{v}}_x \Delta t$ is proposed. It is still necessary to clarify what happens after a proposed configuration is rejected. This is first answered for pair factor potentials between the particles which only depend on the distance between them, $U_{(\{i,j\},T_M)}(\mathbf{r}_i, \mathbf{r}_j) = U_{(\{i,j\},T_M)}(\mathbf{r}_i - \mathbf{r}_j)$, and afterwards generalized.

The new configuration (c', a) is accepted using the consensus principle formulated in Eq. (2.38): Each factor M where $a \in I_M$ determines independently if the displacement of a is accepted or not using Eq. (2.39). (The new configuration is trivially accepted by all factors where $a \notin I_M$.) In a system with N particles and pairwise distance-dependent interactions, the constructed algorithm satisfies maximal global balance if the displacements of a are infinitesimal and if there is a *single* factor $M = (\{a, t\}, T_M)$ which breaks consensus (or, in other words, vetoes the displacement of a). On such a veto, the lifting move

$$(c, a) \to (c, t) \tag{2.47}$$

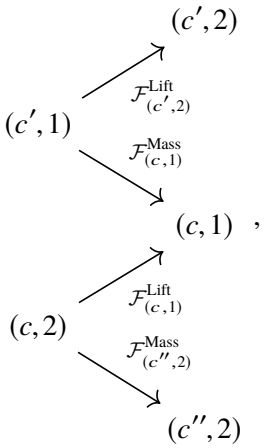
must be introduced. Stated otherwise, if a target particle t rejects the displacement of the active particle a, the configuration c remains unchanged but t becomes active with the same velocity $\hat{\mathbf{v}}_x$. Hence, lifting moves replace rejections and the resulting algorithm is rejection-free on the augmented space. To demonstrate that the constructed algorithm satisfies global balance, that is,

$$\mathcal{F}_{(c,a)} = \mathcal{F}^{\text{Mass}}_{(c,a)} + \mathcal{F}^{\text{Lift}}_{(c,a)} \overset{!}{=} \pi_B(c), \tag{2.48}$$

[2] The velocities introduced here are not derived from the mechanical equations of motions and their conservation laws.

the flow into an augmented configuration is split into two parts. $\mathcal{F}^{\text{Mass}}_{(c,a)}$ denotes that mass flow, which is the flow into (c,a) where a was displaced and the new configuration was accepted. $\mathcal{F}^{\text{Lift}}_{(c,a)}$ denotes the lifting flow, i.e., the flow where another particle t was displaced to propose a new configuration which was then rejected by the factor $(\{t,a\},T_M)$.

In the following, the proof that the introduced algorithm obeys global balance for the case of two particles $\{1,2\}$ interacting by the factor potential $U_{(\{1,2\},T_M)}(\mathbf{r}_1-\mathbf{r}_2)$ is presented first. For this case, the time steps (and hence the displacements of the active particle) do not need to be infinitesimal yet. Suppose, without restriction, that at a given time the augmented configuration is $(c=\{\mathbf{r}_1,\mathbf{r}_2\},a=1)$. This configuration can only be reached by two other augmented configurations,



(2.49)

where $c'=\{\mathbf{r}_1-\hat{\mathbf{v}}_x\Delta t,\mathbf{r}_2\}$ and $c''=\{\mathbf{r}_1,\mathbf{r}_2+\hat{\mathbf{v}}_x\Delta t\}$. Since starting from $(c',1)$ the configuration $(c,1)$ is always proposed (that is, the proposal distribution $g[(c',1),(c,1)]=1$), the transition probability $p^{\text{Fact}}(c'\to c)$ is just given by the the acceptance probability $A^{\text{Fact}}(c'\to c)$ of the factorized Metropolis filter [see Eq. (2.36)]. The mass flow into $(c,1)$ is thus given by

$$\mathcal{F}^{\text{Mass}}_{(c,1)}=\pi_B(c')A^{\text{Fact}}(c'\to c)=\pi_B(c)A^{\text{Fact}}(c\to c'). \tag{2.50}$$

The second equality used that the factorized Metropolis filter satisfies a condition similar to the detailed-balance condition as expressed in Eq. (2.37). Since there is only a single factor, the mass flow takes the form

$$\mathcal{F}^{\text{Mass}}_{(c,1)}=\pi_B(c)\exp\left[-\beta\left(U_{(\{1,2\},T_M)}(\mathbf{r}_1-\hat{\mathbf{v}}_x\Delta t-\mathbf{r}_2)-U_{(\{1,2\},T_M)}(\mathbf{r}_1-\mathbf{r}_2)\right)^+\right]. \tag{2.51}$$

The lifting flow into $(c,1)$ depends on the probability that the proposed change from $(c,2)$ to $(c'',2)$ is rejected,

$$\begin{aligned}\mathcal{F}^{\text{Lift}}_{(c,1)}&=\pi_B(c)\left[1-A^{\text{Fact}}(c\to c'')\right]\\&=\pi_B(c)\left[1-\exp\left[-\beta\left(U_{(\{1,2\},T_M)}(\mathbf{r}_1-\mathbf{r}_2-\hat{\mathbf{v}}_x\Delta t)-U_{(\{1,2\},T_M)}(\mathbf{r}_1-\mathbf{r}_2)\right)^+\right]\right].\end{aligned} \tag{2.52}$$

Thus, $\mathcal{F}^{\text{Mass}}_{(c,1)}$ and $\mathcal{F}^{\text{Lift}}_{(c,1)}$ add up to $\pi_B(c)$ and the global-balance condition is satisfied. Since the particles are displaced persistently in one direction and there are no rejections, even maximal global balance is satisfied. The proof relies on the fact that c'' is a translated version of c' and the factor potential is the same in these configurations. (In principle, it must also be shown that the flow out of an augmented configuration equals its Boltzmann weight. However, this is trivially true because there are no rejections.)

Trying to generalize this proof to N particles with the identifiers $\{1,\dots,N\}$ interacting by a single type of pair factor potentials $U_{(\{i,j\},T_M)}(\mathbf{r}_i - \mathbf{r}_j)$ reveals a problem. There is a single mass flow from $(c',1)$ into $(c,1)$. The analogue of Eq. (2.51) includes a *product* over $N-1$ filter factors which contain 1 in their index set (that is, a product over the exponential functions with all factor potentials $U_{(\{1,t\},T_M)}$ with $t \neq 1$). Also, there are $N-1$ lifting flows from (c,t) into $(c,1)$ with $t \neq 1$. Each of these flows depends on the probability that the factor $(\{1,t\},T_M)$ vetoes the proposed change of configuration by displacing t starting from the configuration (c,t). The total lifting flow into $(c,1)$ therefore contains a *sum* over the filter factors which include 1 in their index set. Global balance is restored by considering infinitesimal time steps because the factorized Metropolis filter then includes a sum as already seen in the infinitesimal limit of the factorized Metropolis filter in Eq. (2.44).

Infinitesimal time steps $\mathrm{d}t$ correspond to infinitesimal displacements $\mathrm{d}x_a$. The probability that the factor M ($a \in I_M$) breaks consensus due to an infinitesimal change $\mathrm{d}x_a$ of the position $\mathbf{r}_a$ along the x-axis is given by $q_{M,a}\,\mathrm{d}x_a$, where $q_{M,a}$ is the event rate defined in Eq. (2.42). The mass flow into (c,a) depends on the acceptance probability of displacing the particle a by $\mathrm{d}(-x_a)$. Denote by $q_{M,-a}$ the appropriate event rate

$$q_{M,-a} = \beta\left[\partial_{-x_a} U_M(c_M)\right]^+. \tag{2.53}$$

The mass flow is then given by

$$\mathcal{F}^{\text{Mass}}_{(c,a)} = \pi_B(c)\left(1 - \sum_{t=1,t\neq a}^{N} q_{(\{a,t\},T_M),-a}(\mathbf{r}_a,\mathbf{r}_t)\,\mathrm{d}(-x_a)\right). \tag{2.54}$$

The lifting flow into (c,a) is equal to the sum of the flows from all configurations (c,t) with $t \neq a$ where the displacement $\mathrm{d}x_t$ of $\mathbf{r}_t$ was rejected,

$$\mathcal{F}^{\text{Lift}}_{(c,a)} = \pi_B(c) \sum_{t=1,t\neq a}^{N} q_{(\{a,t\},T_M),t}(\mathbf{r}_a,\mathbf{r}_t)\,\mathrm{d}x_t. \tag{2.55}$$

Since the pair factor potentials $U_{(\{a,t\},T_M)}(\mathbf{r}_a - \mathbf{r}_t)$ only depend on the difference of the positions, the probability of a veto is the same when displacing t in one direction or a in the inverse direction,

$$q_{(\{a,t\},T_M),t}(\mathbf{r}_a,\mathbf{r}_t)\,\mathrm{d}x_t = q_{(\{a,t\},T_M),-a}(\mathbf{r}_a,\mathbf{r}_t)\,\mathrm{d}(-x_a). \tag{2.56}$$

Thus, in the infinitesimal limit the lifting flow and the mass flow into (c,a) (and similarly into all augmented configurations) again add up to $\pi_B(c)$. The same argumentation is valid if there are multiple factor types T_M, each of them with a distance-dependent pair factor potential (as will be shown below). The presented proof relies on the infinitesimal limit of the time steps and assumes that there is always a unique factor which vetoes a new infinitesimally changed configuration. For continuous factor potentials, the infinitesimal limit also makes sure that the latter assumption is fulfilled.

The lifting framework can furthermore treat factors M with more than two particles ($|I_M| > 2$) if the corresponding factor potential only depends on the distances of the involved particles [17]. Consider, analogously to Eq. (2.54), the mass flow into (c, a) where a can be part of several factors,

$$\mathcal{F}^{\text{Mass}}_{(c,a)} = \pi_B(c)\left(1 - \sum_{\{M:\, a\in I_M\}} q_{M,-a}(c_M)\,\mathrm{d}(-x_a)\right). \tag{2.57}$$

The lifting flow into (c, a) is

$$\mathcal{F}^{\text{Lift}}_{(c,a)} = \pi_B(c) \sum_{\{M:\, a\in I_M\}} \sum_{\{t\in I_M:\, t\neq a\}} \lambda_{t\to a}\, q_{M,t}(c_M)\,\mathrm{d}x_t. \tag{2.58}$$

Here, $\lambda_{t\to a}$ is the lifting probability to change the active particle from t to a once the factor depending on both particles vetoed the displacement of t. The mass and the lifting flow should add up to $\pi_B(c)$. This is achieved independently for each factor M where $a \in I_M$,

$$q_{M,-a}(c_M)\,\mathrm{d}(-x_a) = \sum_{\{t\in I_M:\, t\neq a\}} \lambda_{t\to a}\, q_{M,t}(c_M)\,\mathrm{d}x_t. \tag{2.59}$$

If the factor M consists of only two particles t and a, and its potential only depends on the distance of these, Eq. (2.56) leads to $\lambda_{t\to a} = 1$, i.e., the presented lifting scheme for such pair factors is also valid if there are more than one different types. Distance-dependent interactions between more than two particles are invariant under the translation of the participants so that the factor derivatives with respect to the different particles sum up to zero,

$$\sum_{k\in I_M} \partial_{x_k} U_M(c_M) = 0. \tag{2.60}$$

The index set I_M is split up into two sets I_M^+ (with positive factor derivatives) and I_M^- (with negative factor derivatives) according to

$$\begin{aligned} k^+ \in I_M^+ \quad &\Leftrightarrow \quad \partial_{x_{k^+}} U_M(c_M) \geq 0, \\ k^- \in I_M^- \quad &\Leftrightarrow \quad \partial_{x_{k^-}} U_M(c_M) < 0. \end{aligned} \tag{2.61}$$

For each $k^+ \in I_M^+$, $q_{M,-k^+} = 0$. That implies $\lambda_{t\to k^+} = 0$, i.e., there are no lifting moves toward k^+. Similarly, for each $k^- \in I_M^-$, $q_{M,k^-} = 0$, i.e., the probability that the factor M breaks consensus vanishes if k^- is displaced. This implies $\lambda_{k^-\to t} = 0$. Lifting moves of a vetoing factor M are always from an active particle in I_M^+ to a target particle in I_M^-. Equation (2.59) should be obeyed for all $a = k^- \in I_M$. Using the definition of the event rate [see Eqs. (2.42) and (2.53)], one gets

$$\forall k^- \in I_M^- : \quad -\partial_{x_{k^-}} U_M(c_M) = \sum_{k^+\in I_M^+} \lambda_{k^+\to k^-}\, \partial_{x_{k^+}} U_M(c_M) \tag{2.62}$$

The differentials can be omitted due to the translational invariance of the potential: displacing a in one direction has the same effect as displacing all other particles t in the factor in the inverse direction.

Also, there is definitely a lifting move if M breaks consensus because the algorithm should be rejection free:

$$\forall k^+ \in I_M^+ : \quad \sum_{k^- \in I_M^-} \lambda_{k^+ \to k^-} = 1. \tag{2.63}$$

The last two equations define the lifting probabilities which are only uniquely defined up to $|I_M| = 3$. For this case there are two possibilities: First, $|I_M^+| = 2$ trivially yields $\lambda_{k^+ \to k^-} = 1$ for every $k^+ \in I_M^+$ and the unique $k^- \in I_M^-$. Second, $|I_M^-| = 2$ allows to lift from the unique $k^+ \in I_M^+$ to two possible particles k_i^- ($i \in \{1, 2\}$) with the probabilities

$$\lambda_{k^+ \to k_i^-} = -\frac{\partial_{x_{k_i^-}} U_M(c_M)}{\partial_{x_{k^+}} U_M(c_M)}. \tag{2.64}$$

For $|I_M| \geq 4$, the choice of the lifting probabilities is not unique. A particular simple choice is

$$\lambda_{k^+ \to k^-} = \frac{|\partial_{x_{k^-}} U_M(c_M)|}{\sum_{t^- \in I_M^-} |\partial_{x_{t^-}} U_M(c_M)|}. \tag{2.65}$$

Further possible choices and their influence on ECMC are discussed in [7].

Up to this point, the lifting framework allowed to perform infinitesimal displacements of an active particle in a fixed direction until a unique factor breaks consensus. On such a veto, a lifting move toward another particle in the factor is performed. The constructed lifting probabilities finally yield an algorithm which obeys maximal global balance. However, the proofs relied on the fact that all factor potentials of the interacting particles are only distance-dependent. Albeit such translational invariant interactions are very relevant in the field of statistical mechanics, the constructed ECMC algorithm lacks generality. If it is not possible to decompose an interaction further into factor potentials with translational symmetry, a general solution is to implement backward lifting moves for this factor. That means, if the non-symmetric factor vetoes the displacement of k, the velocity of k gets inverted [9]. To see that this procedure satisfies global balance, consider a single non-symmetric factor M_{ns}. (This can be done because the parts depending on the factors in the mass and the lifting flow in Eqs. (2.57) and (2.58) compensate independently for each factor.) The lifting flow from this factor into (c, k) is given by

$$\mathcal{F}_{(c,k)}^{\text{Lift},M_{\text{ns}}} = \pi_B(c) q_{M_{\text{ns}},-k}(c_M)\, \mathrm{d}(-x_k), \tag{2.66}$$

which trivially compensates the corresponding part in the mass flow of this factor. However, this scheme only satisfies global balance but not maximal global balance. In a simulation with both kinds of factors, the symmetric ones would be treated by the maximal global balance lifting scheme whereas the non-symmetric ones introduce lifting moves which invert the velocity. To the best of the authors' knowledge, there is no general way to restore maximal global balance for non-symmetric factor potentials at this point in time.

For external potentials a lifting scheme obeying maximal global balance can be constructed [9]. When the external potential rejects a displacement of a particle, instead of inverting the velocity of this particle, the potential is moved forward (or all particles are displaced backwards) until this is rejected by the next active particle. The same scheme can also be used for hard-wall boundary conditions. However, the influence of this scheme on the mixing dynamics of ECMC has not been studied yet. (Periodic boundary conditions, of course, do not need such a treatment.)

It is noteworthy that for systems with only distance-dependent interactions the presented lifting schemes can be easily generalized to a fixed number $n_{\text{ac}} > 1$ of independent active particles, all moving with the *same* velocity [39] (see publication in Appendix B). Factors only involving a single active particle can be treated as described before. A slightly different treatment is needed for factors involving several active particles. In principle, these factors veto a new configuration based on the infinitesimal change of the factor potential due to the simultaneous displacements of the involved active particles. For pairwise distance-dependent interactions between two active particles, the factor potential stays constant and this factor never breaks consensus. Nevertheless, both active particles can be treated independently (that is, the factor can veto a displacement of each active particle). Then, lifting moves toward an already active particle must be rejected. The same scheme is valid for interactions which rely on the distances between more than two particles.

2.5 Event-Driven Implementation

The described algorithm could be implemented in a time-driven manner, for example, for a single active particle: Initially, create a random augmented configuration. The active particle of this configuration gets displaced by $\hat{\mathbf{v}}_x \Delta t$ repeatedly. On each step, all factors involving the active particle check if the new configuration is accepted or rejected independently due to factorized Metropolis filter. If a factor breaks consensus, a new particle gets active according to the presented lifting scheme probabilities. This particle is again displaced repeatedly by $\hat{\mathbf{v}}_x \Delta t$ in the same manner.

In order to ensure an ergodic Markov chain, the direction of motion of the active particles needs to be resampled in a not necessarily random way (for example, in a three dimensional system the alignment axis of the normalized velocity can either be randomly chosen or changed in a cyclic sequence). Only considering the velocity $\hat{\mathbf{v}}_x$ parallel to the x-axis, for example, would result in constant positions with respect to all other Cartesian axes except the x-axis of all particles. In order to avoid bias, this resampling should not happen after a certain number of lifting moves (also called events) but after the sum of time steps equals a given chain duration T^{Chain}. The process in between two changes of the direction of motion is the eponymous *event chain*. (Since the absolute value of the velocity stays constant, one can also end an event chain when the cumulative sum of the sequential displacements equals the chain length ℓ.) The chain duration T^{Chain} can be fixed or chosen according to a distribution and is an essential parameter for the performance of ECMC [13]. A similar procedure must be applied in order to obtain sample averages of an observable (here, the relevant fixed quantity is called sampling duration).

In the time-driven implementation, Δt must be small on the one hand to prevent simultaneous vetoing factors. On the other hand, the simulation is only global balanced for infinitesimal time steps. For these reasons, a non-infinitesimal Δt leads to an inexact algorithm. The check of the consensus of the factors involving the active particle on every time step would also yield a slow simulation, especially for long-range interactions.

The limitations of the time-driven implementation are overcome in an event-driven approach [7, 18, 68]. Assume there is a single active particle a and, since all factors are independent, only a single factor M is considered in the following. Suppose that the position $\mathbf{r}_a$ of a where $a \in I_M$ is repeatedly changed with the velocity $\hat{\mathbf{v}}_x$ parallel to the x-axis in finite time steps Δt, and denote by $c_{M,j\Delta t} = \{\mathbf{r}_a + j\hat{\mathbf{v}}_x \Delta t, \mathbf{r}_{\mathbf{i}} : i \in I_M\}$ the configuration after j time steps. Let $\pi_M^{\text{Event}}(c_{M,i\Delta t})$ be the conditional probability that i successive displacements of a were accepted, followed by a rejected

displacement afterwards. It is given by

$$\pi_M^{\text{Event}}(c_{M,i\Delta t}) = \prod_{j=0}^{i-1} \exp\left(-\beta\Delta U_M^+(c_{M,j\Delta t} \to c_{M,(j+1)\Delta t})\right) \times \left[1 - \exp\left(-\beta\Delta U_M^+(c_{M,i\Delta t} \to c_{M,(i+1)\Delta t})\right)\right]. \tag{2.67}$$

In the limit of infinitesimal time steps $\mathrm{d}t$ (which implies infinitesimal positive potential changes $\mathrm{d}U_M^+$), the second term on the right-hand site of Eq. (2.67) becomes $\beta\,\mathrm{d}U_M^+(c_{M,\Delta t_M})$. Here, Δt_M is the time where an event of the factor M and the subsequent lifting move takes place. The exponent of the first term in Eq. (2.67) contains an integral of the factor event rate $q_{M,a}(c_{M,t})$ for displacements of a between 0 and $|\hat{\mathbf{v}}_x|\Delta t_M$ (that is because the product in Eq. (2.67) can be changed to a sum inside of the exponent). This yields the probability density π^{Event} of the positive potential change $\Delta U_M^+(c_M \to c_{M,\Delta t_M})$ of an event taking into account only the increasing parts of the potential [7, 18]:

$$\pi^{\text{Event}}\left(\Delta U_M^+(c_M \to c_{M,\Delta t_M})\right) = \beta \exp\left(-\beta\Delta U_M^+(c_M \to c_{M,\Delta t_M})\right). \tag{2.68}$$

The exponential distribution can be sampled by using inverse transform sampling (see Section 2.1.1) as [44]

$$\beta\Delta U_M^+(c_M \to c_{M,\Delta t_M}) = -\ln[\mathrm{ran}_M(0,1)]. \tag{2.69}$$

Instead of accepting or rejecting the displacement of a at each time step, the order is inverted: First, a positive potential change where the event takes place is sampled. Then, a is continuously displaced along the x-axis (that is, in infinitesimal steps $\hat{\mathbf{v}}_x\,\mathrm{d}t$) until the cumulative positive potential change equals the sampled value. At this point, the next displacement is rejected and a lifting move inside the factor M is carried out. An implicit equation for the candidate event time Δt_M (and hence the corresponding displacement η_M of a along the x-axis) depends on the integrated factor event rate as

$$\underbrace{\beta\Delta U_M^+(c_M \to c_{M,\Delta t_M})}_{\text{Sampled by Eq. (2.69).}} = \int_0^{\Delta t_M} \mathrm{d}t|\hat{\mathbf{v}}_x|\, q_{M,a}(\{\mathbf{r}_a + \hat{\mathbf{v}}_x t, \mathbf{r}_i : i \in I_M\}) \tag{2.70}$$

Equation (2.70) can be viewed as a having an exponentially distributed random value of "energy" it can spend. Every displacement increasing the factor potential costs while every displacement which lowers the factor potential is free. When no energy is left, a lifting move takes place (see Fig. 2.4).

For several factors M involving a, each factor provides a candidate event time Δt_M and the next event takes place at

$$\Delta t^{\text{Event}} = \min_{\{M:\, a \in I_M\}} \Delta t_M. \tag{2.71}$$

The lifting takes place in the factor which realizes this minimum. For continuous potentials, this factor is uniquely defined. (Although finite-precision arithmetic could yield simultaneous events, these are too rare to be relevant.)

The integration of the factor event rate in Eq. (2.70) can be tedious if it cannot be written in an explicit analytical form. Also, the inversion of the factor potential can be non-trivial [7]. Introducing a bounding factor potential can solve these problems. This must be constructed for every configuration

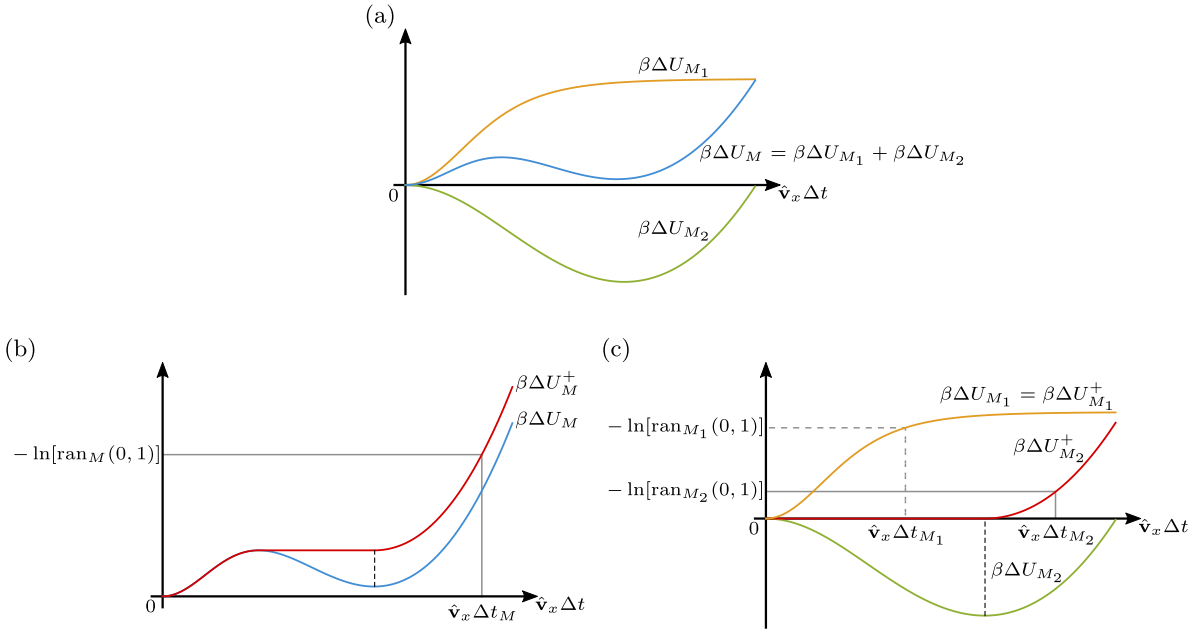


Figure 2.4: Event-driven implementation of ECMC. An active particle a is part of two factors M_i, $i \in \{1,2\}$. Displacing the active particle along the x-axis by $\hat{\mathbf{v}}_x\Delta t$ induces a change ΔU_{M_i} of the factor potentials, and also for the total potential $\Delta U_M = \Delta U_{M_1} + \Delta U_{M_2}$. All plots show the dependence of these changes (scaled by β) on $\hat{\mathbf{v}}_x\Delta t$. (a): The change of the total factor potential ΔU_M (shown in blue) can be split into changes of the factor potentials ΔU_{M_i} (shown in orange and green). (b): The change of the total potential ΔU_M is replaced by the change of the positive total potential ΔU_M^+ where negative increments are replaced by horizontal lines (shown in red, the black dashed line depicts where the change of the factor potential becomes positive again). In order to sample the Boltzmann equation, the active particle needs to be displaced by $\hat{\mathbf{v}}_x\Delta t_M$ where the positive total potential change equals the sampled value $-\ln[\mathrm{ran}_M(0,1)]$. Afterwards, a lifting move takes place in the factor M. (c): Alternatively, the two factors M_i can be treated independently in the same manner as explained for the factor M. This yields two displacements $\hat{\mathbf{v}}_x\Delta t_{M_i}$. The active particle is displaced by the smaller value $\hat{\mathbf{v}}_x\Delta t_{M_1}$ and lifting takes place in the factor M_1.

$c_M = \{\mathbf{r}_i : i \in I_M\}$ so that the bounding factor event rate is an upper bound to the event rate of the real factor potential for every configuration $c'_{M,\Delta t} = \{\mathbf{r}_a + \hat{\mathbf{v}}_x\Delta t, \mathbf{r}_i : i \in I_M\}$ with $\Delta t > 0$,

$$q_{M,a}^{\text{Bound}}(c'_{M,\Delta t}) \geq q_{M,a}(c'_{M,\Delta t}). \tag{2.72}$$

The candidate event time Δt_M is then obtained by integrating the bounding factor event rate. However, the resulting event has to be confirmed with the probability $q_{M,a}(c'_{M,\Delta t_M})/q_{M,a}^{\text{Bound}}(c'_{M,\Delta t_M})$ [69, 70]. Only on confirmed events, a lifting move takes place. (This process is usually called thinning.) A particularly simple bounding factor event rate is a constant value, for example used in the cell-veto algorithm.

The velocity of the active particle was, so far, chosen normalized and parallel to the x-axis. This simplifies the notation as the factor event rate then depends on the partial derivative of the factor potential along the x-axis. For general velocities $\mathbf{v}$, this derivative is replaced by a directional derivative. Moreover, the factor $|\mathbf{v}|$ in the integration of the event rate in Eq. (2.70) not only ensures correct units but can also become numerically important.

With general velocities $\mathbf{v}$, the presented ECMC algorithm remains valid as long as the target particle of a lifting move continues to move with the same velocity. It is possible to construct a global-balanced algorithm for distance-dependent factor potentials which changes $\mathbf{v}$ during a lifting move [8, 71]. Here, the normal component of the velocity with respect to the distance vector is reflected. However, this version showed a reduced performance for hard-disk systems as it obeys detailed balance. Hence, the version described in this thesis seems more promising (and is the one implemented in JF-V1.0).

Note that, in principle, the magnitude of the velocity could change between two different event chains. This effectively influences the distribution of chain durations. As long as the resulting chain is ergodic, this can just have, as mentioned, an influence on the performance of ECMC [13].

2.6 Cell-Veto Algorithm

In principle, the event-driven implementation of ECMC relies on the event-rate integration and inversion for all factors M where $a \in I_M$. For a system with N particles interacting by a long-range pairwise potential (like the Coulomb potential), this gives $\mathcal{O}(N)$ integrations and inversions per event. The overall complexity of ECMC can be reduced to $\mathcal{O}(1)$ by using the cell-veto algorithm [29].

For concreteness, consider the pair factors $(\{i,j\},T_M)$ for all $N(N-1)/2$ pairs where the factor potential U_M is only singular at $\mathbf{r}_i = \mathbf{r}_j$. (The described algorithm can be generalized in a straightforward manner). The system is divided into cells $\mathcal{C}_i$. A cell-occupancy system then maps each particle onto the cell it belongs to and vice versa. For each pair of non-neighbored disjoint cells, the cell-event rate $q_M^{\text{Cell}}(\mathcal{C}_a,\mathcal{C}_t)$ is computed so that (the active particle moves again with velocity $\hat{\mathbf{v}}_x$)

$$q_M^{\text{Cell}}(\mathcal{C}_a,\mathcal{C}_t) \geq q_{M,a}(\mathbf{r}_a,\mathbf{r}_t), \quad \forall \mathbf{r}_a \in \mathcal{C}_a, \mathbf{r}_t \in \mathcal{C}_t. \tag{2.73}$$

It gives an upper bound for the event rate of a single factor $(\{a,t\},T_M)$ with the active particle located in the active cell $\mathcal{C}_a$ and the target particle in the target cell $\mathcal{C}_t$. Neighbored target cells must be excluded in the discussed case because the cell-event rate would diverge. The cell-event rates are tabulated in advance of the ECMC simulation together with the total cell-event rates

$$Q_{T_M}^{\text{Cell}}(\mathcal{C}_a) = \sum_{\mathcal{C}_t} q_M^{\text{Cell}}(\mathcal{C}_a,\mathcal{C}_t), \tag{2.74}$$

where the sum runs over all non-neighbored disjoint cells $\mathcal{C}_t$ of $\mathcal{C}_a$. The total cell-event rate $Q_M^{\text{Cell}}(\mathcal{C}_a)$ is an upper bound for the summed event rates of the *set* of factors of type T_M where the active particle is located in $\mathcal{C}_a$ and interacts pairwise with hypothetical target particles located in each cell $\mathcal{C}_t$. (For a distance-dependent factor potential, the total cell-event rate does not depend on the active cell $\mathcal{C}_a$.) Remaining factors of type T_M are the factors $(\{a,t\},T_M)$ where t is in $\mathcal{C}_a$ or in a neighbor cell (nearby particles), and surplus particles in cells containing more than one particle (see Fig. 2.5).

The candidate event time $\Delta t_{T_M}^{\text{Cell}}$ of the next cell-veto event is obtained according to Eq. (2.70),

$$\Delta t_{T_M}^{\text{Cell}} = \frac{-\ln[\text{ran}(0,1)]}{\beta\, Q_{T_M}^{\text{Cell}}(\mathcal{C}_a)|\hat{\mathbf{v}}_x|}, \tag{2.75}$$

and competes with the event times of the factors concerning nearby and surplus particles, as well as factors concerning distinct factor types. If $\Delta t_{T_M}^{\text{Cell}}$ is the smallest event time, there are two possible

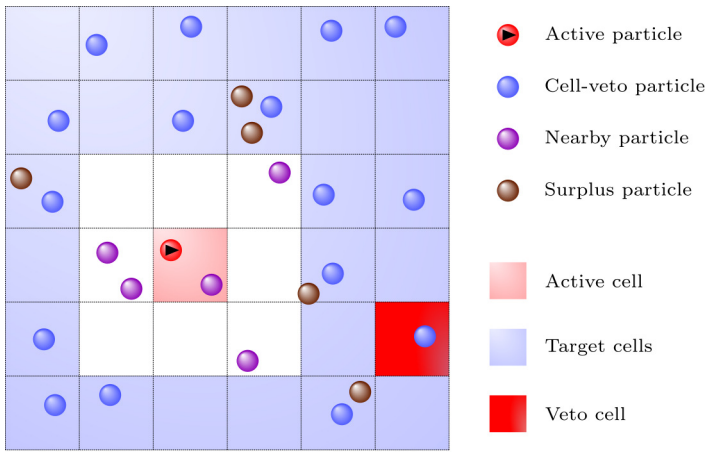


Figure 2.5: Cell-veto algorithm for $N = 27$ particles all interacting by $(\{i,j\},T_M)$ pair factors with the factor potential U_M only singular at $\mathbf{r}_i = \mathbf{r}_j$. The system is divided into 36 cells and a map from the cells onto its occupants is constructed. The total cell-event rate is an upper bound for the summed event rates of the interactions between the active particle and a hypothetical target particle in each target cell. Thus, it can treat the 16 factors between the single active particle and the cell-veto particles. The factors for the 5 nearby and 5 surplus particles must be considered separately. If the cell-veto event yields the smallest candidate event time in competition with the ones of the nearby and surplus factors, Walker's algorithm determines the veto cell. The cell-veto particle in this cell gets active if the cell-veto event is confirmed.

outcomes: In the first outcome, the displacement $\hat{\mathbf{v}}_x \Delta t^{\text{Cell}}_{T_M}$ moves a outside of its cell $\mathcal{C}_a$. Then, a is placed on the boundary to the neighbored cell in the moving direction and the cell-occupancy system is updated (as the cell-event rates are only valid upper bounds as long as the active particle is in its active cell). In the second outcome, the displacement leaves the active particle in $\mathcal{C}_a$. In this case, the veto cell has to be determined (i.e., the vetoing factor has to be extracted from the set of factors treated by the cell-veto algorithm).

The cell-veto event corresponds to the cell $\mathcal{C}_t$ with probability $\propto q^{\text{Cell}}_M(\mathcal{C}_a,\mathcal{C}_t)$. This discrete sampling problem is solved in $\mathcal{O}(1)$ by Walker's algorithm [29, 72]. The cell-event rates $q^{\text{Cell}}_M(\mathcal{C}_a,\mathcal{C}_t)$ are cut and reassembled for each possible active cell $\mathcal{C}_a$ so that at most two cut rates add up to the mean cell-veto rate $Q^{\text{Cell}}_{\text{mean}}(\mathcal{C}_a) = Q^{\text{Cell}}_{T_M}(\mathcal{C}_a)/N_{\text{Cell}}$. Here, N_{Cell} is the number of target cells included in the sum in Eq. (2.74). (This reassembling is done in advance of the ECMC simulation when the cell-event rates are tabulated.) Let the reassembled rates be numbered by $\{1,\dots,N_{\text{Cell}}\}$. The veto cell $\mathcal{C}_t$ is then determined with probability $\propto q^{\text{Cell}}_M(\mathcal{C}_a,\mathcal{C}_t)$ by sampling an uniformly distributed integer between 1 and N_{Cell} and afterwards an uniformly distributed real number between 0 and $Q^{\text{Cell}}_{\text{mean}}$. The former determines the relevant reassembled rate, the latter chooses one of the at most two contributing rates and therefore the veto cell $\mathcal{C}_t$.

The target particle t is the particle present in the sampled veto cell $\mathcal{C}_t$. The event is finally confirmed with the probability $q_{(\{a,t\},T_M),a}(\{\mathbf{r}_a + \hat{\mathbf{v}}_x \Delta t^{\text{Cell}}_{T_M}, \mathbf{r}_t\})/q^{\text{Cell}}_M(\mathcal{C}_a,\mathcal{C}_t)$. Only on a confirmed event, t becomes active. Otherwise, a is displaced by $\hat{\mathbf{v}}_x \Delta t^{\text{Cell}}_{T_M}$ and new events are computed. (If there is no target particle in $\mathcal{C}_t$, the event is trivially rejected.) The cell-veto algorithm treats a set of factors and never performs the integration of the event rate in Eq. (2.70) for its factors. Instead, only a singular factor event rate computation is performed per (confirmed or unconfirmed) event. This also implies, that the total Boltzmann potential remains unknown during the evolution of the Markov chain [7].

The performance of the cell-veto algorithm depends on the total cell-event rates $Q^{\text{Cell}}_{T_M}(\mathcal{C}_a)$ which themselves depend on the range of the treated potential [29]. Also, the total cell-event rates typically get larger for smaller cell sizes. At the same time, the size of the cells should be not too large so that the number of nearby and surplus particles stays small. For an optimal cell size, these two effects must be balanced out. The cell-veto algorithm increases the needed memory and the cell-event rates have to be computed in advance of the ECMC simulation. The latter can be, depending on the factor potential, a time-consuming task itself but has to be done only once per cell-occupancy system and factor type T_M. During the ECMC simulation, the cell-occupancy system must be updated also after events which are not cell-veto events.

2.7 Event-Chain Monte Carlo Simulation of the Hard-Sphere Model

The previous sections introduced the mathematical formulation of ECMC and also explained the implementation of an ECMC simulation. The first basic ingredient is the factorized Metropolis filter which formulates a consensus principle of factors. A proposed configuration in the Markov chain is accepted or rejected independently by all factors. The lifting framework augments the configurations by marking particles as active. The active particles uniquely determine the proposed configurations. This concept allows to move the active particles persistently with a velocity $\hat{\mathbf{v}}_x$. For infinitesimal time steps, it was shown that ECMC satisfies maximal global balance for continuous distance-dependent interactions if lifting moves are introduced. A lifting move changes the active particle when a factor breaks consensus to another particle taking part in this factor, and therefore replaces the rejection of a proposed configuration. The event-driven implementation overcomes the limitations of a time-driven one by inverting the usual order of the simulation. Instead of first proposing a new configuration and then checking if all factors accept it, the candidate event times at which consensus is broken are sampled for each factor independently. The active particles are moved with their velocity for the smallest candidate event time and the corresponding factor determines the lifting move. In order to construct an ergodic Markov chain, the direction of the velocity needs to be resampled after the cumulative sum of the actually traveled times equals a previously chosen chain duration T^{Chain}. The cell-veto algorithm allows to sample the event with the smallest candidate event time and its factor for a set of factors in $\mathcal{O}(1)$ operations, which finally reduces the overall complexity of ECMC.

To conclude this chapter, the presented ECMC algorithm will be applied to the hard-sphere model, introduced in Section 2.2, in two dimensions. Here, a single hard disk is active at any time. Initially, an augmented configuration is chosen, that is, the disks are placed in the system so that there are no overlaps between them, and an active disk with a normalized velocity $\hat{\mathbf{v}}$ either parallel to the x- or the y-axis is chosen randomly. Moreover, the chain duration T^{Chain} of the first event-chain is sampled (in the most simple version, it is just chosen constant). The active disk is displaced straight along its moving direction until it collides with another disk. The latter then becomes active itself and is displaced in the same direction until the next collision with yet another disk. This procedure is repeated until the cumulative sum of all traveled times equals T^{Chain} (see Fig. 2.6). After the end of an event chain, a new active disk, a new direction of motion, and a new chain duration T^{Chain} are sampled and the new event chain is started. Similarly, the configurations are sampled after a fixed sampling duration. The ECMC simulation can be optimized by varying the distribution of T^{Chain} and by implementing a cell-occupancy system to efficiently compute the collision distances of neighbored disks [73].

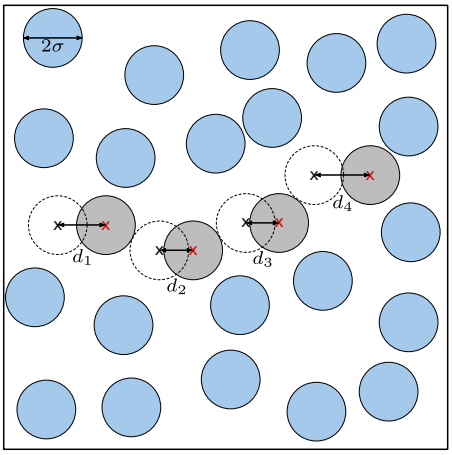


Figure 2.6: A single event chain of an ECMC simulation for $N = 25$ hard disks with radius σ (shown in blue) at the packing fraction $\eta = 0.35$. The initially active disk (the most left gray disk) is displaced with velocity $\hat{\mathbf{v}}_x$ along the positive x-axis for the time $t_1 = d_1/|\hat{\mathbf{v}}_x|$ until it collides with another disk. The latter becomes active instead and is moved with the same velocity for the time $t_2 = d_2/|\hat{\mathbf{v}}_x|$ where it collides itself with another disk. This procedure is repeated until the cumulative sum of the traveled times equals the chain duration, $T^{\text{Chain}} = (d_1 + d_2 + d_3 + d_4)/|\hat{\mathbf{v}}_x|$. The initial positions of the disks which were moved during the event chain are depicted as dotted circles.

As mentioned, ECMC simulations first determined the nature of the phase transition of hard disks in 2011 [15]. In the liquid phase, both the positional and the orientational correlations decay exponentially, whereas in the solid phase there is long-range orientational order and quasi-long-range positional order (i.e., the positional correlations decay as a power law). It was shown that the transition from the liquid to the solid phase in hard disks proceeds in two steps via an intermediate hexatic phase. Here, the positional correlations decay exponentially but there is quasi-long-range orientational order. The hexatic-solid transition is continuous while the liquid-hexatic transition is, in fact, a first order phase transition. The latter was observed, among other phenomena, as a region of the packing fractions where both phases coexist.

CHAPTER 3

Architecture of the Application

The overall architecture of the JF application adopts the mediator design pattern [40]. The different elements (Python classes) do not connect to each other directly but delegate their interaction to a *mediator*. This design maximizes modularity, simplifies changes of the different elements and allows for future extensions. The mediator also contains the main iteration loop over the legs of ECMC's continuous-time evolution of a piecewise non-interacting system, where each leg is interrupted by an event (see Section 3.1). It is therefore the central element of the JF application and introduced at the end of this chapter in Section 3.7. Beforehand, an introduction to the general purposes of the different elements is followed by a discussion of their implementations and properties in the Sections 3.2–3.7.

In the event-driven implementation of ECMC, a non-active particle with identifier i is simply described by its position $\mathbf{r}_i$. The time-dependent position $\mathbf{r}_i(t)$ of an active particle is represented by a time-sliced position $\mathbf{r}_i(t_i)$ at the time stamp t_i and a velocity $\mathbf{v}_i$, i.e.,

$$\mathbf{r}_i(t) = \begin{cases} \mathbf{r}_i & \text{if } \mathbf{v}_i = 0 \text{ (non-active particle),} \\ \mathbf{r}_i(t_i) + (t - t_i)\mathbf{v}_i & \text{else (active particle).} \end{cases} \tag{3.1}$$

In JF, the global state contains all information of Eq. (3.1) and is stored in the *state handler* (see Section 3.2). The global physical state stores the time-sliced position $\mathbf{r}_i$ of all particles whereas the global lifting state stores the time stamps and velocities of only the active particles.

The event times Δt^{Event} for factors involving an active particle, introduced in Section 2.5, are computed with respect to its time stamp. JF introduces a total simulation time. An event taking place at t^{Event} time-slices the active particle, that is, the time stamp is updated as $t_i \to t^{\text{Event}}$ and the position as $\mathbf{r}_i(t_i) \to \mathbf{r}_i(t^{\text{Event}}) = \mathbf{r}_i(t_i) + (t^{\text{Event}} - t_i)\,\mathbf{v}_i$; hence, $\Delta t^{\text{Event}} = t^{\text{Event}} - t_i$. A lifting move transfers the velocity and the time stamp from the active to a non-active particle participating in the vetoing factor (where the non-active particle is determined by using the lifting probabilities introduced in Section 2.4). ECMC requires time-slicing only for particles whose velocity change. However, JF-V1.0 also performs time-slicing after unconfirmed events although the out-state continues the non-interacting continuous-time evolution of the in-state for the sake of simplicity and consistency in the main iteration loop of ECMC (this is going to be discussed in detail in Section 3.7).

Events in JF consist of a candidate event time and an out-state. An event is computed based on an in-state in the *event handlers* (see Section 3.3). For a factor M with the index set I_M containing a single active particle $a \in I_M$, the in-state consists of the global physical state of all particles in I_M

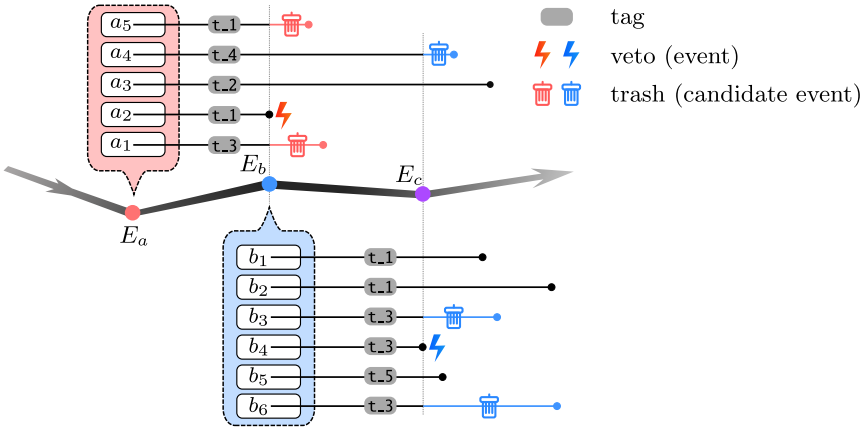


Figure 3.1: The ECMC algorithm implements the time evolution of a piecewise non-interacting system. For each leg, all relevant factors determine a candidate event time at which it breaks consensus, that is, at which time the factor does not accept the continuation of the non-interacting time evolution. The out-state of the event of the factor with the earliest event time determines the time evolution of the next leg. (For example, the leg $E_a \to E_b$ is terminated by the event of the factor a_2.) In JF-V1.0, at the end of each leg candidate events with certain tags have to be trashed (tags `t_1` and `t_3` at E_b) while others are maintained (tags `t_2` and `t_4` at E_b). Also, new candidate events are activated, i.e., their candidate event time is computed (tags `t_1`, `t_3` and `t_5` at E_b). JF also introduces the concept of pseudo-factors. These complement the factors used in the factorized Boltzmann distribution (and whose events are therefore needed for global balance) and allow to formulate ECMC entirely in events. This figure has been published in [39] (see publication in Appendix B).

and the global lifting state of a. From this information, the candidate event time where M breaks consensus and an event takes place is computed [see Eq. (2.70)]. (The in-state may contain more information than that, as explained in Section 3.2. The factor configuration is just the minimum content.) In the out-state, the particles are time-sliced and the velocity is transferred if necessary.

In order to sample the Boltzmann distribution, candidate event times for all factors involving a are stored and compared in the *scheduler* of JF (see Section 3.4). Only the out-state of the event with the smallest candidate event time is committed to the global state.

Factor events interrupt the time evolution of a piecewise non-interacting system so that the Boltzmann distribution is sampled (see Fig. 3.1). JF introduces pseudo-factors which are independent of potentials and have no incidence on the global-balance condition. An event of a pseudo-factor determines, e.g., the end of an event chain after the chain duration T^{Chain} with its candidate event time and specifies a new velocity and active particle in its out-state. Also, events are created whenever an active particle crosses a cell boundary of a cell-occupancy system to keep it consistent with the global state (see Fig. 3.2). Even the start and the end of a run are formulated as events of pseudo-factors.

After the out-state of an event was committed to the global state, some candidate events become invalid while others remain valid. A change of an active particle, e.g., requires the novel computation of events for all factors including the new active particle. The events for the old active particle should be trashed in the scheduler. The event for the end of the run, however, remains valid and has to be maintained. The *activator* (see Section 3.5) is responsible for the maintain/trash decision of events stored in the scheduler, as well as for the identification of event handlers which need to compute new events. It also determines the particle identifiers of the in-state needed by the respective event handlers to compute their event. For this, it may use internal states, e.g., a cell-occupancy system.

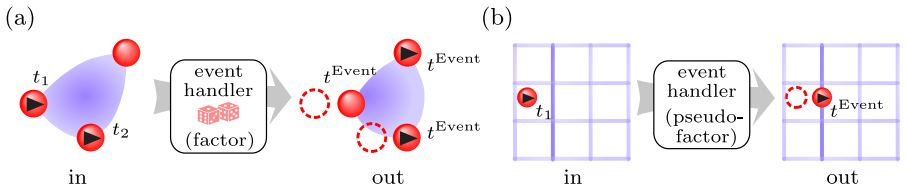


Figure 3.2: Factors and pseudo-factors in the JF application. (a): In- and out-state of an event of a factor involving three particles. The in-state contains two active particles that are time-sliced at times t_1 and t_2, respectively. In the out-state, a lifting move was carried out and all particles are time-sliced at the event time t^{Event}. (b): In- and out-state of an event of a pseudo-factor. The active particle, time-sliced at t_1 in the in-state, is time-sliced at t^{Event} in the out-state. Here, the active particle is placed on the next cell boundary in its direction of motion. A modified version of this figure has been published in [39] (see publication in Appendix B).

The *input-output handler* breaks up into a single *input handler* and possibly several *output handlers* (see Section 3.6). The former is responsible for the initialization of the global state, for example, by reading it from a file or creating it randomly. The output handlers create output during the run of the application. Examples include the sample values of observables or even a complete state of the run. The output handlers are run after certain events of pseudo-factors.

The factors and pseudo-factors of ECMC are statistically independent. The event handlers can thus compute events *in parallel* in their own processes, which connect to the main process containing the main iteration loop of ECMC in the mediator. After all candidate event times were sent to the main process, the mediator requests the out-state from the event handler that created the event with the smallest event time. Depending on the number of available processors, this out-state could have already been computed in advance. (Imagine, e.g., two event handlers computing events on a sufficient number of processors. The first event handler finishes the computation of the candidate event time a long time before the second one. The mediator must wait for all candidate event times so that the first event handler can already compute its out-state, whether or not it belongs to the event with the shortest event time.) JF-V1.0 uses multiple processes instead of threads because of the Python global interpreter lock. The data exchange between the different processes results in a currently slower multi-process version of the application than the single-process implementation. Nevertheless, the underlying idea of treating the independent factors and pseudo-factors in parallel seems important enough that the multi-process version remains a part of the application and this thesis (see Section 3.7).

ECMC is valid for $n_{\mathrm{ac}} > 1$ independent active particles. In a single process, each active particle leads to events sortable by their event times. Sequentially committing these events to the global state (followed by trashing/maintaining events and the computation of new events) yields a correct algorithm if lifting moves to already active particles are rejected. A parallel version of ECMC for a small number of independent active particles, $1 \ll n_{\mathrm{ac}} \ll N$, aims to commit n_{pr} events of different active particles *at once* to the shared global state. Here, conflicts arise if the result disagrees with the single-process version. As a possible solution, each active particle suggests a set of events (computed in parallel) with candidate event times smaller than a global time lock, which is chosen so that the number of conflicts remains small. Also, nearby active particles, which create conflicts with a higher probability, are treated together. Conflicts are resolved centrally and the remaining events are committed to the global state. The implementation of these ideas is one of the main challenges for future versions of JF. The remaining part of this thesis assumes $n_{\mathrm{ac}} = 1$ which is the only case fully implemented in JF-V1.0 (and also the choice of most ECMC applications so far).

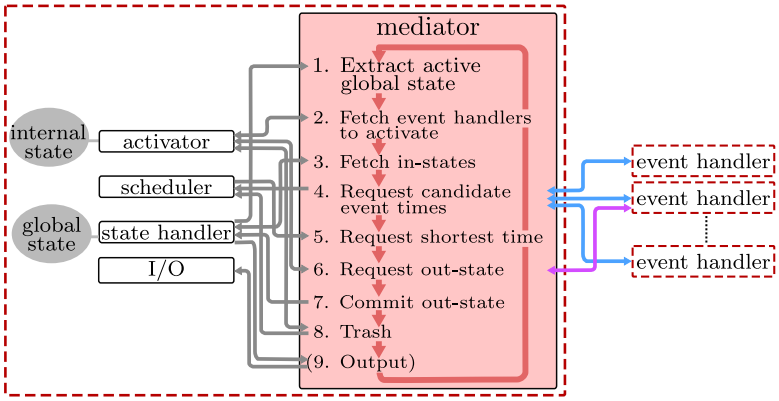


Figure 3.3: The architecture of the JF application is built on the mediator design pattern. The mediator contains the main iteration loop of ECMC over the legs of a non-interacting continuous-time evolution which takes the simulated system from one event to the next (for example, from event E_a to E_b in Fig. 3.1). This happens in the shown nine steps. During these, the different elements of JF do not connect to each other directly but only via the mediator. In the multi-process version of the application, the event handlers have their own iteration loop in a separate process which then interacts with the main process of the mediator. This figure has been published in [39] (see publication in Appendix B).

3.1 Main Iteration Loop of Event-Chain Monte Carlo

The mediator exists in two versions: The `SingleProcessMediator` class runs the application in a single process whereas the `MultiProcessMediator` class communicates with event handlers that have their own autonomous iteration loop in their own separate process. Both implement the `run` method which contains an iteration loop over the legs of the continuous-time evolution of ECMC. Here, both versions share the same conceptual stages. The `run` method is called at the beginning of the simulation by the executable `run.py` script of the JF application (see Section 5.2).

To get from one event to the next during one leg, the mediator goes through nine stages (see Fig. 3.3). In step 1, the active global state (that is, the part of the global state that appears in the global lifting state) is extracted from the state handler. This information is sent to the activator which can then return the event handler and the corresponding in-state identifiers to the mediator in step 2. The identifiers are transformed into in-states using the state handler in step 3. In step 4, the candidate event times are requested from the event handlers which were returned by the activator earlier (the request is accompanied by the corresponding in-states). Returned candidate event times are pushed into the scheduler so that, in step 5, the shortest event time can be received from the scheduler. In step 6, the out-state is obtained from the event handler responsible for the event with the shortest event time, which is then committed to the global state in the state handler in step 7. In step 8, the activator determines the events that should be trashed in the scheduler given the latest committed event. In the final (optional) step 9, output handlers in the input-output handler are run so that, e.g., an observable is sampled. This step is only executed after events of certain event handlers have been committed.

Before the implementation of these steps in the two versions of the mediator can be understood in detail in Section 3.7, it is necessary to understand the different elements of JF and how they perform their actions.

3.2 State Handler

The state handler of JF stores the global state which is split up into the global physical state and the global lifting state. The communication with the mediator requires five public Python methods to be implemented. This is enforced by the abstract `StateHandler` class. The concrete class inheriting from this class has to overwrite the methods given below which makes sure that the mediator can use this class as a state handler.[1] Throughout this thesis, the methods of the different abstract classes are given like they are defined in JF, that is, including the obligatory `self` argument and type hints[2]. The abstract methods of the state handler and their actions are listed in the following:

1. `initialize(self, global_state: Any) -> None`:
   Initializes the state handler with the given global state. This method is used at the beginning of a run of the application before the start of the main iteration loop of ECMC. The argument is created by the input handler. (This method returns nothing which is specified by the return type `None`.)

2. `extract_global_state(self) -> Any`:
   Extracts the full global state so that it can be sent to an output handler which uses it, e.g., to sample an observable (called in step 9 in Fig. 3.3).

3. `extract_active_global_state(self) -> Any`:
   Extracts the part of the global state which appears in the global lifting state (called in step 1 in Fig. 3.3). The output of this method is sent to the activator via the mediator so that it can identify event handlers which need to compute new events.

4. `extract_from_global_state(self, identifier: Any) -> Any`:
   Extracts the part of the global state that corresponds to the given identifier (called in step 3 in Fig. 3.3). This method gets called by the mediator to convert the in-state identifiers received from the activator into in-states which are then sent to the event handlers.

5. `insert_into_global_state(self, extracted_global_state: Any) -> None`:
   Inserts the extracted part of the global state (with changed data) into the stored global state (called in step 7 in Fig. 3.3). This method commits the out-state of an event to the global state.

The return type `Any` of the extraction methods 2.–4. expresses that the inheriting class determines not only the objects used within the state handler to store the global state, but also which objects are used to broadcast parts of it. Other elements of JF must be able to handle the data format (e.g., an event handler must be able to compute a candidate event time based on the object(s) the `extract_from_global_state` method returned). The specific implementation of the state handler similarly determines the possible identifiers that can be arguments of method 4.

The event handlers modify the part of the global state they received. They transform the in-state into the out-state which gets an argument of the `insert_into_global_state` method. The output

[1] Since Python is a dynamically-typed language and also uses duck typing, it is possible to use any class as the state handler in the mediator. However, if the given methods are not defined, an exception will be thrown. The abstract class is thus only a way of broadcasting the methods one needs to define.

[2] Type hints were introduced in PEP 484.

handler and the activator, in contrast, just read the data they received via the mediator from the methods 2. and 3., respectively. Hence, only the `extract_from_global_state` method must copy data stored in the global state (this requires much less copying of data than copying the full global state because an in-state only consists of a small part of the global state).

JF-V1.0 uses a tree state handler which introduces the concept of *composite point objects* and *point masses*. The latter correspond to the particles interacting via the factor potentials and whose positions are sampled corresponding to the Boltzmann distribution using the ECMC algorithm. The former represent barycenters of certain collections of point masses (say, molecules and parts of molecules). For example, two oppositely charged point masses could be connected to a single composite point object which then describes the barycenter of the dipole formed by the two point masses.

Composite point objects and their point masses are connected by storing them in a tree described by nodes. Each node stores its unique parent node (if there is one) and a (possibly empty) list of child nodes. Moreover, each node contains a `Particle` object which stores its time-sliced position. Point masses, which are stored on the leaf nodes of the tree, can additionally have charges stored in a map (a Python dictionary) from the charge's name onto the corresponding value. A tree may have an arbitrary number of levels. The tree's inner nodes then describe the barycenter of parts of the molecule, and the root node that of the entire molecule. Several composite point objects are stored in the tree state handler by storing their root nodes in a list. This list therefore stores the entire global physical state.

The unique identifier of a certain composite point object or point mass is given by a tuple containing a variable number of integers. The first entry of the tuple determines the relevant root node. The (optional) following integers repeatedly refer to a node in the lists of child nodes and traverse down the tree of the root node. The length of the identifier tuple thus corresponds to the level of the node within its tree. The global lifting state is stored using these identifiers. Here, a map from the identifier of each active composite point object or point mass onto its velocity and time stamp is established. (The global lifting state is not stored within the trees because the number of nodes with non-zero velocity is typically small in ECMC.) The velocities inside a composite point object are kept consistent. This means that a moving point mass induces a movement of the composite point object it belongs to (with a velocity equal to the arithmetic mean of the velocities of all its children). The global lifting state therefore stores independent and induced velocities (see Fig. 3.4a–b).

Introducing composite point objects allows for collective motions of entire composite point objects in a straightforward way (see the simulation in Section 6.2.4). Furthermore, internal states (which are kept in the activator) may refer to positions of composite point objects and not only to positions of point masses. A cell-occupancy system can, for example, map identifiers of non-leaf nodes onto the cells their position belongs to (see Fig. 3.4c). By this, the cell-veto algorithm is implementable for an effective interaction between composite point objects. Assume, e.g., that the cell-occupancy system tracks barycenters of dipoles. It is possible to use this in order to set up the cell-veto algorithm for the effective Coulomb interaction between the dipoles, which consists of the sum of the Coulomb potentials between the point masses in different dipoles (see the simulation in Section 6.2.3).

The tree state handler combines the information of the global physical and lifting state into units to communicate with the other elements of JF via the mediator. A unit stores all information of a composite point object or point mass, that is, the identifier, position, charge, velocity, and time stamp. During its construction, the position and the velocity are only optionally copied. All other elements can access units or even modify them (for units which copied the data of the global state). The stored identifier of a unit allows to integrate them into the global state. Thus, units are a common packaging format in JF-V1.0.

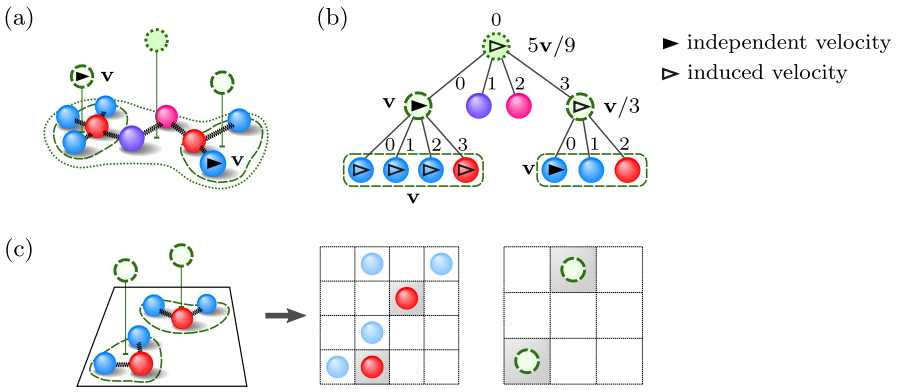


Figure 3.4: Tree representation of composite point objects and point masses in JF-V1.0. (a): Different parts of a molecule consisting of several atoms (shown as solid spheres in different colors which indicate a charge) can be combined and described by the barycenters of the atoms belonging to the parts (shown as the green dotted and dashed lines/circles). (b): The tree state handler of JF-V1.0 stores the atoms as point masses, and the barycenters of parts of the molecule as composite point objects and connects them in a tree. Each node in the tree stores a position and, for the leaf nodes which correspond to the point masses, a charge. Also, each node has a unique identifier which can appear in the global lifting state that assigns a velocity and a time stamp to the node. A velocity is either independent or induced. (c): Different cell-occupancy systems can store point masses with a certain charge (left cell-occupancy system) or even composite point objects (right cell-occupancy system). A modified version of this figure has been published in [39] (see publication in Appendix B).

As a design principle, event handlers keep the time-slicing of composite point objects and their point masses consistent. The returned objects of the `extract_from_global_state` method (which are the in-states of the event handlers) are thus branches of the trees stored in the state handler. A branch of an identifier includes information about the corresponding node together with the ancestors and descendants of the node. The method constructs a temporary copy of the immutable node structure of the state handler where each node contains a unit with copied positions and velocities (see Fig. 3.5). To distinguish nodes containing `Particle` objects, which never leave the tree state handler, and nodes containing units, which are shared among the application, the latter will be called cnodes.

For the sake of consistency, the `extract_active_global_state` and `extract_global_state` method also return branches. The former method constructs the minimal number of branches so that all identifiers appearing in the global lifting state are represented. Each cnode in these branches contains an active unit (a unit where the velocity is not `None`). The latter method simply constructs branches of cnodes for each root node whereby the positions and velocities are not copied. Since this method is called only rarely (in most simulations once per a unique sampling duration), the construction of branches for the full global state does not have a great impact on the run-time of the application (especially for the simulations presented in Chapter 6 where the global states are relatively small). Nevertheless, future improvements of the tree state handler might modify the data format returned by this method if there is an impact in later uses of the application.

In JF-V1.0, the tree state handler contains dynamic information (like the positions and velocities) and static information in the form of the charges of the point masses. The charge maps are not copied during the construction of the units, but they store a reference to the maps instead. Another possibility would be to store the charges in a shared storage which can be accessed by all elements. The current way is, however, more consistent as the charges are close to the other information of the point masses.

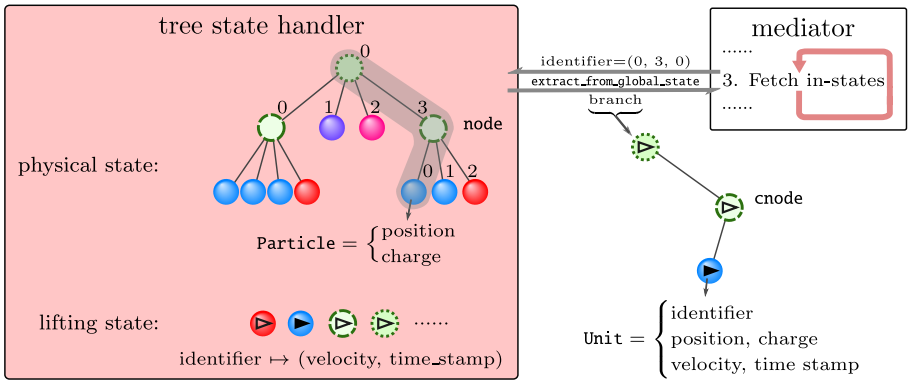


Figure 3.5: Storage of the composite point object presented in Fig. 3.4b in the tree state handler of JF-V1.0. The global physical state is stored in a tree built of nodes. A node is connected to its parent and child nodes, which implies a unique identifier of each node. Moreover, each node contains a `Particle` object that stores the position of the node. For leaf nodes, which correspond to point masses, the `Particle` object can further store a charge. The global lifting state stores a map from an identifier of a node containing an active point mass or composite point object onto its velocity and time stamp. On a call of the `extract_from_global_state` method, the tree state handler constructs a branch for the node corresponding to the identifier given as the argument of the method. The branch also includes all ancestor and descendant nodes. Each node then contains a unit which combines all information from the global physical and global lifting state. This figure has been published in [39] (see publication in Appendix B).

The tree state handler is specifically implemented for simulations of particle systems. The methods of the abstract state handler and their usage in the mediator are, however, not a subject to this restriction. The simulation of, e.g., spin models requires a new state handler and a new input handler to initialize it. Also, the physics of the model has to be implemented, i.e., the event handlers for appearing factors and pseudo-factors, as well as output handlers to sample observables. (The activator of JF-V1.0 needs only a limited amount of additions as discussed in Section 3.5). This shows that the modular design of JF allows for a straightforward extension to other models.

3.3 Event Handler

During a run of the application, there is only a unique state handler as there is only a single global state to manage. Similarly, there is only a unique scheduler, activator, input-output handler (which contains, however, several output handlers) and mediator (see Fig. 3.3). This does not apply to the event handlers. Usually, events of multiple factors and pseudo-factors have to be computed, which is the responsibility of many different event handlers.

Event handlers that realize a single (pseudo-)factor receive the full in-state on a request of the candidate event time. The in-state is stored internally and also used to compute the out-state of the event. In contrast, event handlers that compute events for a set of (pseudo-)factors receive only the part of the in-state which is needed to compute the candidate event time on a request thereof. That is because the element of the set that triggers the event is not yet known at the time of the request. The event handler may return supplementary arguments together with the candidate event time, which are used by the mediator to construct the missing part of the in-state of the vetoing (pseudo-)factor in the set (see Section 3.7). This part then accompanies the out-state request (see, for example, the

discussion about the event handler in Section 4.1.4 that computes cell-veto events, which concern a set of factors as discussed in Section 2.6).

The necessity to allow for both types of event handlers implies a comparably complex definition of the two methods responsible for computing the candidate event time and the out-state of an event, respectively, in the abstract `EventHandler` class:

1. `send_event_time(self, *in_state: Sequence[Any]) -> Union[float, Tuple[float, Sequence[Any]]]`:
   Computes the candidate event time based on the optional in-state (called in step 4 in Fig. 3.3). If there is one, the mediator will transmit it as a single sequence (the type of the members of the sequence depends on the state handler). Returns either only the candidate event time or a tuple additionally containing arguments in a sequence that are used by the mediator to construct the arguments of the `send_out_state` method.

2. `send_out_state(self, *args: Any) -> Any`:
   Computes and returns the out-state of the event (called in step 6 in Fig. 3.3). If the arguments of the `send_event_time` method were only a part of the in-state, the remaining part is an argument of this method.

The different event handlers implemented in JF-V1.0 are described in Chapter 4.

3.4 Scheduler

The scheduler keeps track of the candidate event times and their associated event handlers. For this, the abstract `Scheduler` class enforces the implementation of three methods:

1. `push_event(self, time: float, event_handler: EventHandler) -> None`:
   Stores the candidate event time computed by the given event handler (called in step 4 in Fig. 3.3).

2. `get_succeeding_event(self) -> EventHandler`:
   Returns the event handler which computed the smallest candidate event time currently stored in the scheduler (called in step 5 in Fig. 3.3). The returned event handler and its candidate event time should not be trashed as the activator is responsible for determining events to trash.

3. `trash_event(self, event_handler: EventHandler) -> None`:
   Trashes the candidate event time which was computed by the given event handler (called in step 8 in Fig. 3.3).

JF-V1.0 uses a heap scheduler, that is, the tuples of the candidate event times and the corresponding event handlers are stored and sorted using a heap queue.[3] Inserting a new element into a queue of n elements is $\mathcal{O}(\log_2 n)$ and the event handler corresponding to the shortest candidate event time is returned in $\mathcal{O}(1)$ [74]. The (not yet optimal) trashing of a certain event handler consists of traversing the queue until it is found. Future versions will implement lazy deletion instead (i.e., rather than searching through the queue, an already trashed event handler is discarded when it comes out of the queue naturally within the call of the `get_succeeding_event` method). For the simulations presented in Chapter 6, n does not get very large which implies that the scheduler does not have a great impact on the performance of the application.

[3] For this, the Python `heapq` module is used.

3.5 Activator

At the beginning of each leg of the piecewise non-interacting time evolution of ECMC, the activator of JF provides to the mediator the new event handlers that have to compute a candidate event time. (Note that, according to the mediator design pattern, no data flows directly between the event handlers and the activator, i.e., the activator does not call any methods of the event handlers. Nevertheless, the activator initially obtains the references of the event handlers and manages them.) The activator also determines, possibly with the help of internal states, the in-state identifiers for each of the returned event handlers. These are, after converting them to in-states using the `extract_from_global_state` method of the state handler, the arguments of the `send_event_time` method. Finally, the activator yields the event handlers whose events should be trashed after the out-state of an event of a preceding event handler has been committed to the global state. The abstract `Activator` class thus defines four methods:

1. `initialize(self, extracted_global_state: Any) -> None`:
   Initializes the internal states of the activator using the given extracted global state (the type of the argument depends on the state handler). The internal states should only use the identifiers of the global state and not duplicate the data stored within the state handler. This method is used at the beginning of a run of the application before the start of the main iteration loop of ECMC.

2. `get_event_handlers_to_run(self, extracted_active_global_state: Any, preceding_event_handler: EventHandler) -> Mapping[EventHandler, Union[Sequence[Any], None]]`:
   Returns a map from all event handlers onto their in-state identifiers that have to compute candidate events in the next iteration of the main iteration loop of ECMC (called in step 2 in Fig. 3.3). The arguments consist of the part of the global state that appears in the global lifting state (returned by the `extract_active_global_state` method of the state handler) and the preceding event handler that committed an out-state to the global state. The in-state identifiers should be a sequence of identifiers whose type depends on the used state handler. If the event handler does not require an in-state, the corresponding identifier is `None`.

3. `get_trashable_events(self, preceding_event_handler: EventHandler) -> Sequence[EventHandler]`:
   Returns a sequence of event handlers that are passed to the `trash_event` method of the scheduler in order to trash its candidate event time, given the preceding event handler that committed an out-state to the global state (called in step 8 in Fig. 3.3). The preceding event handler should be included in the returned sequence as the scheduler did not delete it when determining it. By this, only the activator is responsible for trashing events.

4. `get_info_internal_state(self, event_handler_asking: EventHandler, identifier_in_internal_state: Any) -> Any`:
   Returns the global state identifier stored in an internal state that is associated with the given internal state identifier. The type of the latter depends on the implementation of the relevant internal state (called in step 6 in Fig. 3.3). The type of the returned identifier depends on the used state handler. In JF-V1.0, this method is used to construct the missing part for the out-state request of an event handler realizing a set of factors (e.g., the cell-veto event handler in Section 4.1.4). The "asking" event handler then determines the relevant internal state.

JF-V1.0 uses a tag activator with tags providing a finer distinction than the factor types T_M (i.e., a certain factor type can be treated by several event handlers with different tags). Event handlers and their events are internally associated with tags using *taggers*. The tag activator's operations depend on the interdependence of these tags: The tag of the preceding event handler that committed an out-state to the global state determines the tags of event handlers which compute candidate event times next. Similarly, the event handlers whose events are trashed in the scheduler are determined (see Fig. 3.1).

Taggers are also responsible for the determination of the in-state identifiers of its event handlers. A tagger receives, on initialization, a tag (if no tag is given, instead the name of the class is used), a single instance of an event handler, and a number of desired event handlers. It then uses the Python `deepcopy` method to create as many identical event-handler copies as needed. The in-state identifiers for the `send_event_time` method of its event handlers are generated in the tagger's `yield_identifiers_send_event_time` method, which has the extracted active global state as the only argument. (This method is called in the activator's `get_event_handlers_to_run` method and the argument `extracted_active_global_state` is passed through.) A tagger may use an internal state to generate the in-state identifiers. This associates, at the same time, an event handler with an internal state, which is used for the implementation of the `get_info_internal_state` method.

The number of event handlers inside a tagger should meet the maximum number of events associated to the tagger's tag simultaneously stored in the scheduler. That is because the corresponding event handlers might need to compute the out-state of a stored event and can thus not compute new events. Note that event handlers and their candidate events may be referred to by tags, although they do not have a tag property (like the taggers). For the sake of completeness, all taggers implemented in JF-V1.0 and a description of how they create the in-state identifiers are listed in the Appendix A.

On initialization, a tagger additionally receives a list of tags that it creates and trashes, respectively. The lists of all taggers are used to construct the create and trash maps from taggers onto lists of taggers. At the beginning of a run, the tag activator further establishes a map from the event handlers onto their taggers. Also, it sets up an internal pool of event handlers that stores for each tagger which event handlers are currently running (i.e., an event of it is stored in the scheduler) and not running. On a call of the `get_event_handlers_to_run` method, the preceding event handler that committed an out-state to the global state (given as an argument of the method) is first mapped onto the corresponding tagger. Afterwards, the create map yields the taggers whose `yield_identifiers_send_event_time` method is iterated. The tag activator then extracts a not-running event handler of the corresponding tagger for each of the generated in-state identifiers in order to construct the return value. Here, all returned event handlers become running in the internal pool of the tag activator. A call of the `get_trashable_events` method is answered by returning all running event handlers of taggers that appear in the trashing map of the tagger corresponding to the preceding event handler. Here, all returned event handlers become not-running again (see Fig. 3.6).

For the very first call of the `get_event_handlers_to_run` method in a run of the application, no preceding event handler can be provided because no event handler committed an out-state to the global state yet. Hence, the tag activator initially returns a start-of-run event handler (together with the in-state identifiers generated by its tagger). This is an event handler that inherits from the abstract `StartOfRunEventHandler` class (see Section 4.2.4).

The concept of creating certain taggers can be overruled by the concept of deactivated taggers. These do not generate any in-state identifiers and thus, its event handlers are never returned in the `get_event_handlers_to_run` method. At the beginning of the same method, taggers get activated and deactivated based on tag lists which were set on initialization of the taggers.

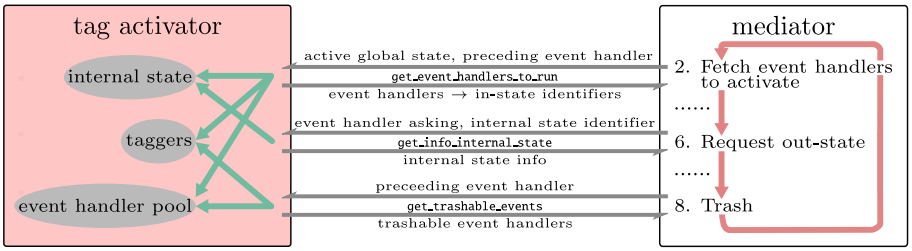


Figure 3.6: Interaction of the mediator with the tag activator of JF-V1.0. The tag activator consists of three parts: The taggers associate event handlers and their events with a tag and generate in-state identifiers for its event handlers. The interdependence of tags yields the relevant taggers used in the methods of the tag activator. In order to generate the in-state identifiers, furthermore internal states can be used. Finally, the tag activator manages a pool of event handlers. Event handlers are either running, that is, an event of them is stored in the scheduler, or not running. Only not-running event handlers can compute new events and only running event handlers can be trashed. This figure has been published in [39] (see publication in Appendix B).

It is noteworthy that, since the generation of the in-state identifiers of event handlers based on the active global state is delegated to the taggers, the tag activator can be used with any state handler. A new state handler, for example treating spin models, requires only the implementation of new taggers. For this, the abstract `Tagger` class is designed so that the inheriting class usually only needs to implement the `yield_identifiers_send_event_time` method (besides the obligatory `__init__` method). Other properties of a tagger, e.g., how to activate and deactivate a tagger, are already implemented in the abstract class. Also, a new state handler leads to new internal states. The internal states currently implemented in JF-V1.0 for the tree state handler are described in the following.

3.5.1 Internal State — Cell-Occupancy System

The activator of JF may contain internal states that support the determination of in-state identifiers for event handlers which should compute candidate events next. Such internal states are initialized at the beginning of the run of the application with the extracted full global state returned by the `extract_global_state` method of the state handler (see the `initialize` method of the activator). However, they should not double up on the information available in the state handler. This means that, for the case of the tree state handler, the internal states do not store the positions of point masses or composite objects again but rather contain additional information about them.

In JF-V1.0, the internal states consist of cell-occupancy systems. These are specifically implemented for the tree state handler and associate identifiers of point masses or composite point objects to cells. Cell-occupancy systems are typically used for two things: On the one hand, they allow for book-keeping, e.g., the efficient determination of point masses within a certain distance of a given active point mass. On the other hand, they can be used for cell-based bounding potentials [see Eq. (2.73)].

Each cell-occupancy system is associated with a cell system that inherits from the abstract `Cells` class. Cell systems consist in a regular grid of cells whereby each cell is referred to by a unique identifier. These identifiers are iterable via the `yield_cells` method. Each cell possesses a lower and upper bound position in each direction accessible via the `cell_min` and `cell_max` methods. Hence, a position can be uniquely associated to a cell using the `position_to_cell` method. The lifting framework used in ECMC often requires to determine a successor cell of a given cell in a certain

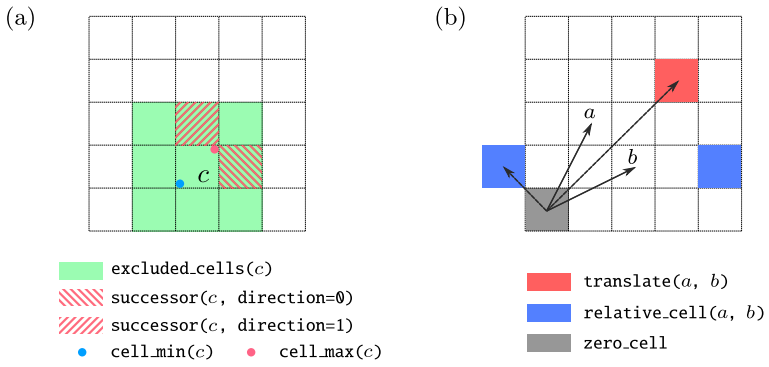


Figure 3.7: Methods of the cell systems of JF-V1.0. (a): The abstract `Cells` class introduces a unique cell identifier for each cell. For a given cell c, the cell system gives access to the lower and upper bound positions in this cell via the `cell_min` and `cell_max` methods, respectively. The `successor` method returns the successor cell in the horizontal, indexed by 0, and vertical direction, indexed by 1, respectively. On initialization, each cell can furthermore define a set of excluded cells that are returned by the `excluded_cells` method. In this figure, the neighbored cells of c are excluded. (b): The abstract `PeriodicCells` class additionally assumes periodic boundary conditions and translational invariance. The periodic-cell system's origin is given by the `zero_cell` property. The `relative_cell` determines the relative blue cell that has the same cell separation with the `zero_cell` as the cells a and b. (The two blue cells are identical due to the periodic boundaries). The `translate` method returns the red cell which has the same distance to a as the cell b to the `zero_cell`. This figure has been published in [39] (see publication in Appendix B).

direction. This is implemented in the `successor` method. (As the event handlers of JF-V1.0 are only implemented for velocities parallel to one of the Cartesian coordinate axes, see Chapter 4, the same restriction appears in this method.) Finally, it might be necessary to define a set of excluded cells for each cell that is returned by the `excluded_cells` method (this appears, for example, in the cell-veto algorithm in Section 2.6 where neighbored cells of the active cell were excluded). The excluded cells should be set on initialization of the inheriting class (see Fig. 3.7a).

The abstract `PeriodicCells` class (which itself inherits from the `Cells` class) enhances the cell system to a periodic and translational invariant cell system. The `zero_cell` property corresponds to the cell located at the origin of the coordinate system. The `relative_cell` method receives a cell and reference cell and returns a relative cell. Here, the method established an equivalence between the pair consisting of the origin and relative cell and the pair of cells that appeared in the argument of the method. The opposite of this is the `translate` method (see Fig. 3.7b).

JF-V1.0 implements periodic cuboid cells which receive, on initialization, three arguments: first, the number of cells per coordinate axis that, together with the system size, determines the size of a single cuboid cell; second, the origin of the cell system (i.e., the minimum corner of the `zero_cell`); and third, the number of neighbor layers that should be excluded for each cell, i.e., the number of cells excluded in each direction of a given cell. This information is enough to properly implement all methods of the abstract `PeriodicCells` class, whereby it simply references each cell by an integer.

The described cell systems are used by cell-occupancy systems. In JF-V1.0, each cell-occupancy system also has a `cell_level` property. This determines the length of the global state identifiers of the tree state handler (which are tuples of integers) that are actually mapped onto the given cell system. By this, both the identifiers of point masses and composite point objects may be considered. The `cell_level` property introduces, however, the restriction that the identifiers, which should actually

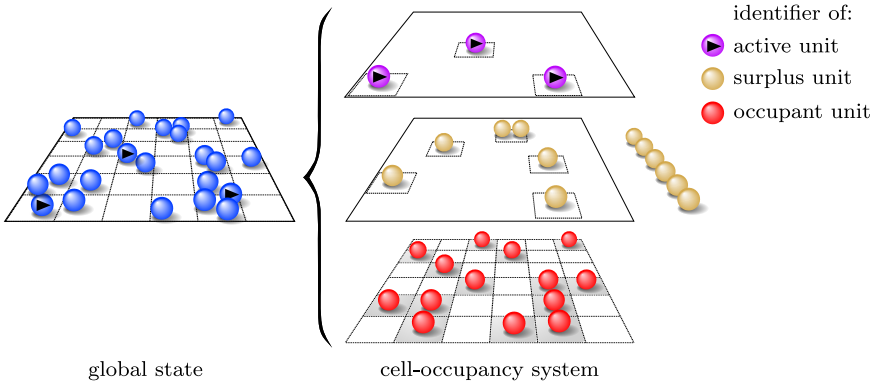


Figure 3.8: A cell-occupancy system which serves as an internal state of the activator. The cell-occupancy system associates global state identifiers of units of a certain length to a cell of an underlying cell system. Only a single unit's identifier is stored as an occupant per cell. Identifiers of surplus units are accounted for differently and can be iterated over. The same is true for identifiers of active units and their active cells. A modified version of this figure has been published in [39] (see publication in Appendix B).

be stored by the cell-occupancy system, must have the same length. For the example of mapping point masses onto the corresponding cells, the cell-occupancy system can thus only be used for global states where each tree in the tree state handler has the same height (i.e., the same number of levels).

As mentioned, the cell-occupancy system receives the full global state before the main iteration loop of ECMC is entered. The type of the global state depends on the state handler. For the case of the tree state handler, it is given by a sequence of branches of all root cnodes. The cell-occupancy system then only stores identifiers of relevant units, that is, the ones with the correct length. Each cell of the underlying cell system is associated to a single identifier for which the corresponding unit's position lies within the cell (accessible via the `__getitem__` method). Of course, there can be more than one relevant unit within a certain cell. Such additional occupants are stored as surplus identifiers and should be generated by the `yield_surplus` method. Finally, relevant units of the cell-occupancy system may be active. These are extracted from both the stored occupant and surplus storages. Instead, the identifiers of relevant active units are generated together with the corresponding active cells by the `yield_active_cells` method (see Fig. 3.8).

The discussed methods are used by taggers. For example, a tagger can easily generate in-state identifiers for event handlers that realize surplus factors of the cell-veto algorithm using the `yield_surplus` method and the identifier of a single active unit. Similarly, using the `excluded_cells` of the underlying cell system, in-state identifiers for event handlers realizing excluded factors can be generated. More taggers using a cell-occupancy system are explained in the aforementioned Appendix A.

The cell-occupancy system stored in the activator and the global state stored in the state handler have to be kept consistent. This means that the time-sliced position $\mathbf{r}(t_i)$ and the time-dependent position $\mathbf{r}(t)$ of an active unit in Eq. (3.1) belong to the same cell. The tag activator achieves this by calling the `update` method of the cell-occupancy system at the beginning of the `get_event_handlers_to_run` method with the extracted active global state as the only argument. A pseudo-factor triggers a cell-boundary event if the active relevant unit of the cell-occupancy system crosses a cell boundary. In the out-state of this event, the active unit has the minimal position within the successor cell in its direction of motion (see Section 4.2.1). By this, the corresponding active cell is properly updated.

JF-V1.0 implements the `SingleActiveCellOccupancy` class. Here, only a single relevant unit and its active cell are kept within its private properties. The occupant identifiers are stored as a list where each member corresponds to a single cell. An internal Python dictionary furthermore maps from the cell identifiers onto the surplus identifiers. In the `update` method, the class first checks if the stored active unit identifier is still active. If not, it is inserted either into the occupant list or into the surplus map. Afterwards, the new active unit identifier is extracted from the internal storage of occupants and surplus identifiers. If the active unit identifier remained the same, just the active cell is recomputed.

It is noteworthy that the `SingleActiveCellOccupancy` class can further decide if identifiers of units are stored based on charges. For this, a charge name may be set on initialization. The cell-occupancy system then only stores identifiers of units with both the correct length and a non-zero charge of the corresponding name. Also, a single run of the application can feature several different instances of this class, whereby each instance can rely on different cell systems and consider different charges and identifier lengths (see Fig. 3.4c). Each cell-occupancy systems that stores an active unit requires the creation of separate cell-boundary events.

3.6 Input-Output Handler

The input-output handler decomposes into a single obligatory input handler and several optional output handlers. Both are defined by an abstract class so that the input-output handler can use them properly.

3.6.1 Input Handler

The input handler is responsible for the initialization of the global state in the state handler. For this, the abstract `InputHandler` class defines a single method:

1. `read(self) -> Any`:
   Creates the initial global state that is passed, via the mediator, to the `initialize` method of the state handler. The returned object(s) should be suited for the respective state handler. This method is called by the input-output handler in an equally named method which itself gets called at the beginning of a run of the application before the start of the main iteration loop of ECMC.

JF-V1.0 implements two input handlers for the tree state handler. Both of these only initialize the global physical state (that is, the positions and charges that are stored in a list of trees) but not the global lifting state. The initial event chain is instead started via the out-state of the start-of-run event handler (see Section 4.2.4).

The first input handler reads the initial global physical state out of a file in the Protein Data Bank (pdb) file format [75]. The pdb file is parsed by the Python MDAnalysis package [76, 77]. (This package is not a part of the Python standard library. Note, however, that this package only has to be installed when this input handler is used.) The second input handler creates a random initial global physical state for the simulations discussed in Chapter 6.

Both input handlers only allow to create a global physical state of N similar composite point objects of height at most two. Thus, every root node (representing composite point objects) has the same number of children that represent point masses. By this, the charges of the point masses are simply initialized by setting them only for the point masses within a single composite point object. The input handlers then assume the same charges for the point masses within all composite point objects.

3.6.2 Output Handler

Every output handler is associated to a file whose name is set on initialization and in which output of a run of the application is created. The output handlers serve many purposes and the abstract `OutputHandler` defines a single very general method:

1. `write(self, *args: Any) -> None`:
   Creates output using the given arguments. This method is called, via the input-output handler, after certain events of pseudo-factors have been committed to the global state (called in step 9 in Fig. 3.3). The argument of the method depends on the event handler that realized the pseudo-factor (see Section 3.7).

Most output handlers are run after an event of a sampling pseudo-factor (see Section 4.2.2). For these cases, the argument of the `write` method consists of the extracted full global state returned by the `extract_global_state` method of the state handler. The output handler then usually samples an observable using this data. This implies that such output handlers are specifically implemented for the composite point objects and point masses that appear in the tree state handler in the given run, and also for the desired observable. Another possibility consists in sampling the entire global physical state and extracting the wanted observables using a separate program. JF-V1.0 implements output handlers for both cases: The observables in the simulations in Chapter 6 are computed in different output handlers. (Here, the observables are separations between certain point masses in different composite point objects.) Another output handler creates a pdb file that contains the entire global physical state, again using the Python MDAnalysis package.

The dumping output handler of JF-V1.0 is used after an event of a corresponding pseudo-factor (see Section 4.2.2). Then, the argument of the `write` method is the mediator object itself. This allows to store the state of an entire run of the application (following the principle of a core/memory dump) because the mediator contains references to all elements of the JF application. All Python objects of the application are serialized and written into a file together with the state of the `random` module. For this, the Python dill package is used [78, 79]. (Again, the Python dill package only has to be installed if this output handler is used). With the help of the created file, the run of the application can be restarted at the state where the output handler was used (which also includes that exactly the same random numbers are drawn in the following). This proves useful for debugging purposes (see Section 5.2).

3.7 Mediator

The single- and multi-process versions of the mediator both contain the main iteration loop of ECMC described in Section 3.1 in their `run` method. For the single-process version, the mediator just calls the introduced public methods of the different elements of JF. A special communication is needed for event handlers that treat a set of (pseudo-)factors. As discussed, these may return additional objects besides the candidate event time in their `send_event_time` method, which should be used to construct the arguments of the `send_out_state` method. Some of these event handlers do not even return any additional objects but need arguments in their `send_out_state` method nevertheless. The mediator design pattern solves this by defining a “get-arguments” method for each class of event

handlers that requires this special treatment.[4] (The get-arguments method may also be defined for an abstract event handler class. If such a method is not implemented for a given class, the method of the base class is used, if it exists.) If a get-arguments methods is implemented for the class of the event handler that should compute an out-state, it is executed with the additional objects returned during the event-time request as the arguments. The returned objects of the get-arguments method then accompany the out-state request.

The same pattern is used to optionally run output handlers in the input-output handler in the final step 9 of the main iteration loop of ECMC (see Section 3.1 and Fig. 3.3). A "mediating" method is defined for certain classes of event handlers that received, on initialization, a name of an output handler (see Section 4.2.2). This method specifies to the input-output handler the relevant output handler and the arguments of the `write` method after an out-state of a corresponding event handler class has been committed to the global state. Both the get-arguments and the mediating methods are implemented in a shared abstract class of the single- and multi-process versions of the mediator.

The multi-process version requires a special communication with the event handlers, which should run their own autonomous iteration loop in a separate process. JF-V1.0 uses, nevertheless, the same event-handler classes for the two versions of the mediator. This is achieved by a monkey-patching technique. The mediator dynamically adds a `run_in_process` method to each event-handler instance in a run of the application. This method contains an iteration loop, which is started in a separate process, that reacts to shared flags set by the mediator. Moreover, the `send_event_time` and the `send_out_state` methods are decorated so that the return values are rather transmitted through a two-way pipe that connects the event-handler process to the main process. Likewise, arguments of the methods are transmitted from the mediator to the event handlers through the pipe. Here, it is important that only the mediator accesses the methods of the event handlers (according to the mediator design pattern). Thus, the redefinitions of methods, which abolish the need for two versions of each event-handler class, do not produce undesired side effects.

For each event handler, the same pipe is used to receive the candidate event time and the out-state. Hence, the multi-process mediator assigns four different stages to the event handlers (`idle`, `event_time_started`, `suspended`, `out_state_started`) in order to distinguish the received objects. Also, the assigned stage determines the flags the mediator can set to either start the `send_event_time` or the `send_out_state` method. In the `idle` stage, the mediator can only set the starting flag. The event handler then waits for the in-state to be transmitted through the pipe. After it was put into the pipe by the mediator, the stage is changed to `event_time_started` at the end of which the event handler places the candidate event time into the pipe. When the mediator has recovered the data, it changes the stage of the event handler to `suspended`. In this stage there are two possibilities: Either the mediator sets again the starting flag which reverts the event handler to the `event_time_started` stage, or it sets a continue flag that lets the event handler compute the out-state and changes the stage to `out_state_started`. After the out-state was recovered from the pipe, the event handler is returned to the `idle` stage (see Fig. 3.9).

The multi-process mediator adjusts the strategy for either suspending an event handler or letting it start the computation of the out-state (before its actual request) to the number of available processors. Here, the computation of event times of all event handlers always has a higher priority. This means

[4] In order to not implement the analogue of a switch statement with extensive type checking of event handlers inside a very long unique get-arguments method, the mediator of JF-V1.0 implements several of the get-arguments methods that contain the class name of the event handlers in their names. At the beginning of a run, the mediator creates a Python dictionary mapping from the existing event handlers onto its get-arguments method, if it is implemented.

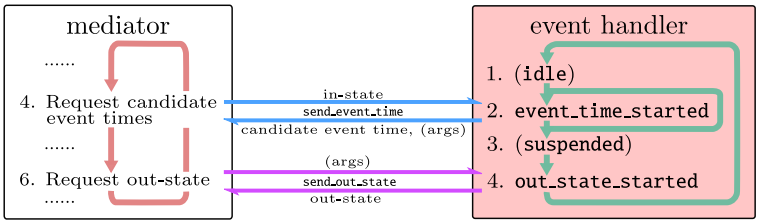


Figure 3.9: The multi-process mediator assigns stages to the event handlers that contain an autonomous iteration loop in a separate process. The arguments and return values of the shown methods are transmitted via a two-way pipe. The mediator can set different shared flags in order to start the computation of a candidate event time or an out-state. The shown stages of the event handler allow the mediator to distinguish the objects received via the pipes, and to determine which flags can currently be set. This figure has been published in [39] (see publication in Appendix B).

that all available processors are first used to compute candidate event times. If processors become free before all times are computed, they are used to compute out-states in advance. In JF-V1.0, the communication via pipes presents a computational bottleneck and slows down the multi-process mediator.

Both versions of the mediator run the main iteration loop of ECMC until an `EndOfRun` exception is raised. In JF-V1.0, this takes place in the mediating method of the abstract end-of-run event handler class (see Section 4.2.4). Thereafter, the `post_run` method of the mediator is executed, which finalizes and closes files in the output handlers and terminates the event handler processes for the multi-process version.

As mentioned, JF-V1.0 also performs time-slicing after unconfirmed events. This means that out-states of events are committed to the global state even when the velocities stored in the global lifting state do not change. This also results in the fact that a whole new iteration in the mediator is started, which includes, in particular, the creation and trashing of events via the activator. Future versions of JF will most probably introduce a smaller feedback loop for unconfirmed events. Here, the event handler computes a candidate event time again if an event is not confirmed in its out-state computation. The time is then directly put into the scheduler and the out-state of the event handler that computed the now smallest candidate event time is requested (in Fig. 3.3, this corresponds to going back from step 6 to step 4 for unconfirmed events).

The presented architecture of the JF application closely mirrors the mathematical formulation of ECMC that was introduced in Section 2. Although the implementations of the current elements of JF-V1.0 are aimed toward all-atom simulations, the definition of these elements is done in a general way and allows for a straightforward extension to other models. The underlying ideas of the application are, of course, not restricted to the Python language. Its popularity and simplicity, however, allows a lot of possible users to understand and use JF, and also to extend it to their purposes.

CHAPTER 4

Event Handlers

The event handlers are an essential part of the JF application. They realize the factors and pseudo-factors of ECMC and compute their events. Events of factors are required by ECMC because they ensure that the flow into a configuration equals its Boltzmann weight. This is not true for events of pseudo-factors that, however, permit JF to represent an entire run of the application in terms of events.

A given event handler does not directly correspond to a given (pseudo-)factor of ECMC because it treats a variety of in-states. Thus, the identifiers of the in-state change for each event computed by an event handler. In contrast, the index set of a (pseudo-)factor consists of a *fixed* set of identifiers. The (pseudo-factor) that is realized in an event handler is therefore only determined after it received the full in-state, and a certain event handler realizes several different (pseudo-)factors during a run of the application. The type of the realized (pseudo-)factor, however, is fixed by the concrete implementation of the `send_event_time` and `send_out_state` methods of an event handler (which uses, e.g., a certain factor potential). It is the task of the activator of JF to determine, on the one hand, the (pseudo-)factors for which an event has to be computed next, and on the other hand, the event handlers that realize those. As discussed in Section 3.5, JF-V1.0 uses tags for both purposes and several tags may treat a single type of (pseudo-)factors.

This chapter summarizes the event handlers implemented in JF-V1.0, and describes how they implement their `send_event_time` and `send_out_state` methods. Section 4.1 introduces the event handlers realizing factors while Section 4.2 introduces the ones realizing pseudo-factors. In both cases, the distinction between event handlers that treat a single (pseudo-)factor and those that treat a set of (pseudo-)factors is crucial as they use a different communication with the mediator (see Section 3.3). The discussed event handlers are used in the simulations of Chapter 6 and are all implemented for the tree state handler (see Section 3.2). This implies that the in-states consist of cnode branches and that the event handlers keep the time-slicing of composite point objects and their point masses consistent in the constructed out-states. Moreover, all event handlers are implemented under consideration of periodic boundary conditions (see Section 5.1).

4.1 Event Handlers for Factors

Event handlers for factors or sets of factors depend on a factor potential acting between several point masses. In order to maximize reusability of the different event handlers, the potentials are not hard coded into the event handlers but are separate elements of JF.

4.1.1 Factor Potentials

In JF-V1.0, there are two kinds of factor potentials: The first kind is inverted, i.e., its event rate is integrated in closed form along the straight line trajectory of the active point mass [see Eq. (2.70)]. The second kind is not inverted (either by choice or because it is not possible). Treating a non-inverted factor potential in an event handler requires an inverted bounding potential, which is used to propose an event time. The associated event is then confirmed or rejected during the computation of the out-state [see the discussion after Eq. (2.72)].

All event handlers for factors in JF-V1.0 concern distance-dependent factor potentials between two or more point masses, one of which is active. A lifting move after an event of the corresponding factor transfers the velocity from the active to another non-active point mass taking part in the factor (see Section 2.4). Factors with non-distance-dependent factor potentials have to invert the velocity during a lifting move and require the implementation of new event handlers.

The potentials of JF-V1.0 are furthermore restricted to velocities of the active point mass that are parallel to one of the Cartesian coordinate axes and in the positive direction, which is the usual choice of ECMC. They only implement their partial derivatives (and similarly the integration of the event rate) along the coordinate axes. It is then sufficient to specify the direction of motion of the active point mass as an integer (e.g., `direction=0` corresponds to the partial derivative with respect to the x-coordinate of the active point mass). General velocities in arbitrary directions necessitate the implementation of new potentials and new event handlers.

Non-inverted factor potentials only implement the `derivative` method (enforced by the abstract `Potential` class) whose arguments are the separation vectors of the involved point masses, the active point mass's direction of motion, and optionally specific charges of all point masses. The method returns the derivative with respect to the active point mass along its direction of motion evaluated at the separation vector. From this, the event handler can determine the factor event rate. (Still, the method returns the derivative because some event handlers require negative derivatives, see Section 4.1.5)

Inverted factor potentials additionally implement the `displacement` method. (This is enforced by the abstract `InvertiblePotential` class that itself inherits from the `Potential` class.) This method receives the same arguments as the `derivative` method, plus a sampled positive potential change [see Eq. (2.69)]. It returns the displacement η_M of the active point mass after which the integrated event rate along its straight line trajectory equals the given positive potential change.

The remaining part of this section lists all potentials between several point masses of JF-V1.0 and shortly describes their implementations of the different methods. It is noteworthy that, in principle, the presented potentials could be also used as an effective interaction between several composite point objects, whereby one of these is active. However, none of the event handlers of JF-V1.0 implement this and all potentials are therefore explained using the terminology of point masses.

4.1.1.1 Inverse-Power-Law Potential

The inverse-power-law potential concerns the separation vector $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ in d-dimensional space between a non-active point mass j and an active point mass i. Thus, the corresponding pair factor is $M = (\{i, j\}, \text{Inverse})$. The potential moreover depends on specific charges c_i and c_j of the point masses,

$$U_{(\{i,j\},\text{Inverse})}(\mathbf{r}_{ij}, c_i, c_j) = c_i c_j k_{\text{Inverse}} \left(\frac{1}{|\mathbf{r}_{ij}|}\right)^{p_{\text{Inverse}}} . \tag{4.1}$$

Here, k_{Inverse} and $p_{\text{Inverse}} > 0$ are parameters that are set on initialization of the potential class. The implementation of the `derivative` method is straightforward by using the analytic form of the partial derivative with respect to a single coordinate of the active particle's position of Eq. (4.1).

The `displacement` method distinguishes between the repulsive ($c_i c_j k_{\text{Inverse}} > 0$) and attractive ($c_i c_j k_{\text{Inverse}} < 0$) cases, respectively. For example, consider the attractive case and the separation vector $\mathbf{r}_{ij} = (x_{ij}, y_{ij}, z_{ij})$ in 3 dimensions. Let the velocity of the active point mass be aligned with the x-axis and $x_{ij} > 0$ initially. The potential then displaces i along the x-axis until the integrated event rate equals the sampled positive potential change ΔU^+. Displacing i until $x_{ij} = 0$ contributes nothing to the integral as the event rate is 0 for all $x_{ij} \geq 0$. Thereafter, it is important that Eq. (4.1) can be algebraically inverted for $|\mathbf{r}_{ij}|$. The potential uses the inversion to compute the value of $|\mathbf{r}_{ij}^{\text{Event}}|$ where the potential equals $U_{(\{i,j\},\text{Inverse})}(\mathbf{r}_{ij} = (0, y_{ij}, z_{ij}), c_i, c_j) + \Delta U^+$. The integrated event rate is equal to the sampled positive potential change at this separation $\mathbf{r}_{ij}^{\text{Event}} = (x_{ij}^{\text{Event}}, y_{ij}, z_{ij})$ because the event rate is greater than zero for all $x_{ij} < 0$. From $|\mathbf{r}_{ij}^{\text{Event}}|$, the potential can determine the value of $x_{ij}^{\text{Event}} < 0$ and finally return the displacement $\eta_M = x_{ij} + |x_{ij}^{\text{Event}}|$. The repulsive case is treated similarly.

The given separation $\mathbf{r}_{ij}$ may be between i and any periodic image of j. The *separate-image* inverse-power-law potential just uses it in its computations without further considering the effect of periodic boundary conditions (for example, the potential does not correct the separation toward the *minimum separation* between i and the closest image of j). By this, different implementations of event handlers can treat the set of interactions between i and several images of j separately.

4.1.1.2 Lennard-Jones Potential

Another separate-image potential of JF-V1.0 implements the Lennard-Jones potential that corresponds to the pair factor $M = (\{i, j\}, \text{LJ})$. However, this potential does not depend on charges of the involved point masses,

$$U_{(\{i,j\},\text{LJ})}(\mathbf{r}_{ij}) = k_{\text{LJ}} \left[\left(\frac{\sigma_{\text{LJ}}}{|\mathbf{r}_{ij}|} \right)^{12} - \left(\frac{\sigma_{\text{LJ}}}{|\mathbf{r}_{ij}|} \right)^{6} \right]. \tag{4.2}$$

Again, k_{LJ} and σ_{LJ} are parameters of the potential that are set on initialization, and $\mathbf{r}_{ij}$ is the separation vector between the point masses whereby i is active. The Lennard-Jones potential class internally uses two instances of the inverse-power-law potential class with accordingly chosen parameters. The `derivative` method returns the sum of the derivatives of the two underlying potentials.

In contrast, the `displacement` method cannot use the underlying potentials as it requires the distinction of more cases due to the minimum of the Lennard-Jones Potential at separations with the norm $r_{ij}^{\text{Min}} = \sigma_{\text{LJ}} 2^{1/6}$. Consider the initial separation $\mathbf{r}_{ij} = (x_{ij}, y_{ij}, z_{ij})$ in 3 dimensions with $x_{ij} > 0$ chosen so that $|\mathbf{r}_{ij}| > r_{ij}^{\text{Min}}$, as well as y_{ij} and z_{ij} small enough so that $|(0, y_{ij}, z_{ij})| < r_{ij}^{\text{Min}}$. (Otherwise the displacement can be computed analogously to the attractive case of the inverse-power-law potential.) The velocity of the active point mass is again parallel to the x-axis. The event rate is zero for all values of $x_{ij} > 0$ where $|\mathbf{r}_{ij}| > r_{ij}^{\text{Min}}$. Hence, the potential first changes x_{ij} to the value where the minimum of the potential is reached. Afterwards, the potential increases until $x_{ij} = 0$ and the height of this potential hill must be compared to the sampled positive potential change. If the latter is larger than the potential hill, the potential decreases again until it reaches its minimum for some value $x_{ij} < 0$, followed by another increase. Both increasing parts of the potential are "climbed" until the cumulative and the sampled positive potential changes are equal and the total displacement of i is returned.

4.1.1.3 Displaced-Even-Power-Law Potential

The displaced-even-power-law potential is a separate-image potential that also depends on the separation vector $\mathbf{r}_{ij}$ in d-dimensional space between two point masses with i active, and is independent of any charges,

$$U_{(\{i,j\},\mathrm{DEPP})}(\mathbf{r}_{ij}) = k_{\mathrm{DEPP}} \left(|\mathbf{r}_{ij}| - r_0 \right)^{p_{\mathrm{DEPP}}}, \tag{4.3}$$

where $k_{\mathrm{DEPP}} > 0$, $p_{\mathrm{DEPP}} \in \{2,4,6,\ldots\}$, and r_0 are parameters that are set on initialization of the potential class. The corresponding pair factor is $M = (\{i,j\},\mathrm{DEPP})$. Both the `derivative` and the `displacement` methods are provided analytically, whereby the implementation of the latter does a similar case distinction as the Lennard-Jones potential.

4.1.1.4 Merged-Image Coulomb Potential

The *merged-image* Coulomb pair potential approximates the sum of the separate-image Coulomb pair potentials between the active point mass i and every periodic image of the non-active point mass j using Ewald sums [80, 81]. These present a reliable technique to treat the long-range Coulomb interaction without introducing inaccuracies by truncating the long-range part (which is done, e.g., when considering only a finite amount of separate-image interactions). This potential is explicitly implemented for the case of periodic boundary conditions. Moreover, this class is restricted to simulations within a cubic box with side length L in three dimensions because JF-V1.0 only implements the Ewald sums for this setting. (The other potentials of this section do not have this restriction and can, for example, also be used in a hypercuboid simulation box in arbitrary dimensions.)

The merged-image Coulomb pair potential depends on the minimum separation vector $\mathbf{r}_{ij,0}$ between the active point mass i and the closest image of the non-active point mass j and, of course, on the charges of the point masses (in appropriate units),

$$U_{(\{i,j\}\mathrm{Coulomb})}(\mathbf{r}_{ij,0}, c_i, c_j) = \sum_{\mathbf{n}\in\mathbb{Z}^3} \frac{c_i c_j}{|\mathbf{r}_{ij,0} + \mathbf{n}L|}. \tag{4.4}$$

The sum (which effectively sums over the periodic images of the non-active point mass) is only conditionally convergent but can be consistently defined in terms of "tin-foil" boundary conditions (that is, the system and all its periodic images are embedded in a medium with an infinite dielectric constant). The idea of the Ewald sums is to screen each charge by a diffuse charge distribution of the opposite sign. The total charge of this cloud exactly cancels the screened charge. This induces, depending on the screening charge distribution, a rapid decay with growing distance of the single charge's electrostatic potential. Here, a Gaussian distribution is assumed for the screening charge clouds. These are themselves compensated by a smoothly varying periodic charge density, which can be represented by a rapidly converging Fourier series. This results in [7, 80, 81]

$$\begin{aligned} U_{(\{i,j\}\mathrm{Coulomb})}(\mathbf{r}_{ij,0}, c_i, c_j) = c_i c_j \Bigg[& \sum_{\mathbf{n}\in\mathbb{Z}^3} \frac{\mathrm{erfc}(\alpha|\mathbf{r}_{ij,0} + \mathbf{n}L|)}{|\mathbf{r}_{ij,0} + \mathbf{n}L|} \\ & + \frac{4\pi}{L^3} \sum_{\mathbf{q}\neq(0,0,0)} \frac{\exp[-\mathbf{q}^2/(4\alpha^2)]}{\mathbf{q}^2} \cos(\mathbf{q}\cdot\mathbf{r}_{ij,0}) \Bigg] + U_{\mathrm{Self}}(\alpha). \end{aligned} \tag{4.5}$$

Here, the Fourier-space sum is over $\mathbf{q} = 2\pi\mathbf{m}/L$ with $\mathbf{m} \in \mathbb{Z}^3$. The self-energy contribution $U_{\text{Self}}(\alpha)$ does not depend on any point masses' positions and is thus irrelevant for the implementation of the `derivative` method. The result of the potential is independent of the tuning parameter $\alpha > 0$, which however influences the convergence of the sums in Eq. (4.5).

The corresponding potential class in JF-V1.0 defines optimized default values for α and the spherical cut-offs of the sums in real and Fourier space. By this, the derivative, which can be analytically calculated from Eq. (4.5), reaches machine precision in the fastest way. (Usually, Ewald sums consider several interacting charges and the optimal values depend on the number of them. Since ECMC treats each Coulomb interaction between two charges separately, the optimal values must only be determined once.) All parameters can be changed on initialization nevertheless. The potential class further accelerates the summation over the Fourier-space components $\mathbf{q} \neq (0,0,0)$ in the derivative by reducing it to a sum over non-negative components and computing the position-independent parts (which especially include the exponentials) before the main iteration loop of ECMC is entered [7].

The merged-image Coulomb potential is not invertible and requires a bounding potential. JF-V1.0 provides a merged-image Coulomb bounding potential that depends on the minimum separation vector $\mathbf{r}_{ij,0}$ and the charges of the two point masses,

$$U_{(\{i,j\},\text{Bounding})}(\mathbf{r}_{ij,0}, c_i, c_j) = k_{\text{Bounding}} \frac{c_i c_j}{|\mathbf{r}_{ij,0}|}. \tag{4.6}$$

The bounding potential shares the singularity of the Coulomb potential at $\mathbf{r}_{ij,0} = (0,0,0)$. However, it should be noted that Eq. (4.6) does not involve any sum over periodic images. The implementation of the `displacement` method requires, nevertheless, the consideration of periodic images of the non-active point mass during the integration of the event rate along the straight-line trajectory of the active point mass. The `derivative` method is again straightforward.

The constant k_{Bounding} is chosen as

$$k_{\text{Bounding}} = \max_{\mathbf{r}\in[-L/2,L/2]^3} \frac{|\mathbf{r}|^3}{x} \partial_x U_{(\{i,j\}\text{Coulomb})}(\mathbf{r}, 1, 1), \tag{4.7}$$

where $\mathbf{r} = (x, y, z)$, so that the Coulomb-potential event rate is bounded. Equation (4.7) can explicitly regard the derivative with respect to x because the derivative of the Coulomb potential is symmetric. The $|\mathbf{r}|^3/x$ term cancels the inverse term that appears in the derivative of the Coulomb potential due to the $1/|\mathbf{r}|$ pole. A constant $k_{\text{Bounding}} \gtrsim 1.5836$ is appropriate for a cubic simulation box.

4.1.1.5 Three-Body Bending Potential

The separate-image three-body bending potential concerns three point masses and the corresponding factor is $M = (\{i, j, k\}, \text{Bending})$. It depends on two separation vectors $\mathbf{r}_{ij}$ and $\mathbf{r}_{jk}$ in d-dimensional space and is given by

$$U_{(\{i,j,k\},\text{Bending})}(\mathbf{r}_{ij}, \mathbf{r}_{jk}) = \frac{1}{2} k_{\text{Bending}} \left(\phi_{\{i,j,k\}}(\mathbf{r}_{ij}, \mathbf{r}_{jk}) - \phi_0\right)^2. \tag{4.8}$$

Here, $\phi_{\{i,j,k\}}(\mathbf{r}_{ij}, \mathbf{r}_{jk})$ denotes the internal angle between the two separation vectors. The constants k_{Bending} and ϕ_0 are set on initialization of the potential.

The three-body bending potential has an important difference compared to the previously described two-body potentials: The separation arguments are the same independently of which of the point masses i, j, and k is active (whereas for the two-body potentials a change of the active point mass induced a change of sign of the separation vector). That is because the bending potential typically describes fluctuations of the bond angle within a water molecule (see the simulation in Section 6.3).

The `derivative` method is provided analytically and returns three derivatives where it assumed a different active point mass in each returned value. However, this potential is not inverted and thus, the `displacement` method is not implemented. In JF-V1.0, an event handler constructs a bounding potential dynamically (see Section 4.1.5). This event handler then also extracts the correct derivative from the list returned by the `derivative` method based on the currently active point mass. This shows that there is, in general, a great freedom in the design of a factor potential. One just needs to take care that the event handlers can handle them and extract the right information.

4.1.1.6 Cell-Based Bounding Potential

A cell-based bounding potential bounds the derivative of a factor potential inside certain cell regions by constants. These are computed analytically or may be even sampled using a Monte Carlo approach in the estimators of JF. These estimators are specifically designed for distance-dependent pair potentials, which it receives on initialization. They then determine upper and lower bounds for the derivative in a given direction of motion for separations inside a hypercuboid with given minimum and maximum corners (within the `derivative_bound` method). Both upper and lower bounds are needed if the factor potential additionally depends on a charge product of the involved point masses (as is the case, e.g., for the merged-image Coulomb potential).

In general, an estimator just compares the derivatives of its potential for different separations within the hypercuboid to obtain the upper and lower bounds. The maximum and minimum values of the derivatives may additionally be corrected by a multiplicative prefactor. Optionally, an empirical bound as an upper limit on the absolute values of the bounds (after the multiplication with the prefactor) can be set. Both the prefactor and the empirical bound are set on initialization of the estimator. The boundary-point and inner-point estimators of JF-V1.0 sample the separations evenly distributed on the boundary of the given hypercuboid or within it, respectively.

Two additional estimators treat the interaction between a charged active point mass and two oppositely charged target point masses that build a dipole with a fixed dipole separation (this implies that these estimators can only be used with potentials that actually consider charges). The derivative of the factor potential (which is typically the merged-image Coulomb potential) is summed for the two possible active-target pairs. A Monte-Carlo estimator then distributes both the separation between the active point mass and the dipole, as well as the dipole-orientation randomly. The dipole-inner-point estimator distributes the separations evenly within the hypercuboid. The dipole orientation is aligned along the direction of the derivative's gradient for each separation. Both estimators are useful for all-atom models where several point masses with compensating charges are combined to a composite point object. A Coulomb interaction is then only present between point masses in different composite point objects (see the simulations in Sections 6.2 and 6.3). Combining the Coulomb pair factors for all point masses in two composite point objects to a larger factor yields factor event rates that decay much faster with distance than the ones for Coulomb pair factors [7]. This is due to the fact that the factor event rate is computed based on a sum of derivatives that compensate each other due to the compensating charges within a composite point object.

The cell-based bounding potential of JF-V1.0 is explicitly implemented for periodic boundary conditions and stores piecewise cell-bounded derivatives for a specific periodic cell system (see Section 3.5.1). (Note that only the activator has access to cell-occupancy systems. The underlying cell systems, however, are used by other objects as well.) Before the main iteration loop of ECMC starts, the cell-based potential iterates over all possible directions of motion and all non-excluded relative cells of the underlying cell system's `zero_cell`. Here, certain relative cells may be excluded because, e.g., the cell-bounded event rate would diverge or be too large. The pair of cells determines the minimum and maximum corners of the hypercuboid of possible separations that are handed to the estimator, together with the considered direction of motion. The returned derivative bounds are then stored internally for all possible relative cells and directions of motion (the lower bound is only computed and stored if a charge should be considered). Only the relative cell is relevant because the bounded factor potential just depends on the separations of the involved point masses.

The `displacement` method receives the direction of motion of the active point mass, the relative cell, optionally the charges, and the sampled positive potential change. The appropriate constant bound on the event rate is easily integrated to compute the displacement. Note that this displacement might actually move the active point mass out of its cell. The cell-based bounding potential does not take this into account. Rather, preempting cell-boundary events render this problem irrelevant (see the discussion in the corresponding event handler in Section 4.1.3). The implementation of the `derivative` method is straightforward as well as it can just use the stored bounds.

The determination of the cell-bounded event rates within the estimators of JF-V1.0 is appropriate for the simulations in Chapter 6 where the number of relative cells remains small. For a large number of relative cells, the estimation of the bounds would need a considerable time before the ECMC simulation even starts. Then, more elaborate methods are required. For example, the cell-bounded event rates can be computed analytically for certain potentials. Note also that the estimators compute the bounds at the start of each run. Another possibility is to run the estimation once and to store the bounds in a file, which is then read in at the beginning of the run by the cell-based bounding potential.

After the implementation of inverted and non-inverted distance-dependent factor potentials has been covered, the event handlers using these are introduced in the following sections. As mentioned before, all of these are specifically implemented for periodic boundary conditions.

4.1.2 Event Handler with an Inverted Factor Potential

The `TwoLeafUnitEventHandler` concerns distance-dependent inverted factor potentials between a pair of point masses, which may additionally depend on specific charges of the involved point masses.[1] It thus realizes a single pair factor $M = (\{i, j\}, T_M)$, whereby the type T_M is fixed by the factor potential that is set on initialization. The index set $\{i, j\}$ is, as mentioned, only fixed after an in-state was received, inside of which only one of the point masses is independent active.

For this event handler, the in-state (which is given by branches of cnodes of the point masses i and j) accompanies the call of the `send_event_time` method. The event handler first extracts the units of the point masses i and j and determines the unique independent active unit i. The velocity $\mathbf{v}$ of the latter yields the direction of motion. Next, an exponentially distributed random number is sampled

[1] An event handler is initialized with a certain potential which may or may not expect charges. Also, the event handler may or may not receive a name of a charge on initialization. On initialization, the event handler checks if the potential is usable. If the name is not `None` but the potential does not expect charges, an exception is thrown. However, if the name is `None` and the potential does expect charges, the event handler usually sets these to 1.

[see Eq. (2.69)]. This, the minimum separation $\mathbf{r}_{ij,0}$ of the two point masses, possibly the charges, and the direction of motion of the active point mass build the arguments of the factor potential's `displacement` method. The factor potential returns the displacement $\eta_M = |\mathbf{v}|\Delta t_M$ of the active point mass after which an event takes place. From this, the event handler computes Δt_M and returns the candidate event time $t_M = t_i + \Delta t_M$, using the time stamp t_i of the active point mass.

The in-state time-sliced to the candidate event time t_M (with positions corrected for periodic boundaries) is stored internally for the subsequent call of the `send_out_state` method. This only takes place if the candidate event time was actually the smallest. To construct the out-state, the velocity and the time stamp of the independent active point mass i are transferred to the non-active point mass j. The time-slicing of all composite point objects present in the branches of i and j are kept consistent.

The event handler explicitly computes the minimum separation $\mathbf{r}_{ij,0}$ between the interacting point masses under consideration of periodic boundary conditions. This is used in the factor potential which is either separate- or merged-image. Treating a pair factor with a separate-image factor potential only by event handlers of this type induces an approximation, as all images of the non-active point mass except the closest one are neglected (which might be a reasonable approximation if the factor potential decays quickly with growing separations). In order to treat other images, new event handlers, which compute the separation accordingly, have to be implemented . This remark is valid for all event handlers described in this chapter because all of them explicitly use the minimum separation vector.

4.1.3 Event Handlers with a Bounding Potential

The event handlers described in this section are used for distance-dependent factor potentials between two point masses which do not provide a `displacement` method (or the method is ignored by choice). Only the `derivative` method is implemented. Similar to the previous section, all the event handlers realize a single pair factor $M = (\{i, j\}, T_M)$ including one independent active point mass.

The `TwoLeafUnitBoundingPotentialEventHandler` uses an inverted bounding potential to compute the candidate event time in a similar manner as described in Section 4.1.2. The bounding potential's event rate must be greater than or equal to the event rate of the factor potential for every displacement along the straight line trajectory of the active point mass [see Eq. (2.72)]. On an out-state request, the event is first confirmed with a probability proportional to the ratio of the event rates of the factor and the bounding potential. Here, the active point mass is time-sliced to the candidate event time and the event rates are computed using the `derivative` method of the two potentials. The velocity and the time stamp of the active point mass are only transferred to the non-active one (while keeping their branches consistent) if the event is confirmed. The out-state is returned to the mediator in both cases.

The `TwoLeafUnitCellBoundingPotentialEventHandler` uses a cell-based bounding potential. As introduced in Section 4.1.1.6, such a bounding potential stores piecewise cell-bounded derivatives for a specific periodic cell system to which the event handler has access. On an event-time request, the event handler first extracts the two point masses from the in-state and localizes them within their respective cells. The cell system's `relative_cell` method then determines the relative cell. This builds, together with an exponentially distributed random number and optionally specific charges of the point masses, all arguments of the cell-based bounding potential's `displacement` method. Similar arguments are used for the `derivative` method. The former method allows to compute the candidate event time, while the latter is used to confirm the event on an out-state request. If required, the out-state is constructed as explained in the previous paragraph.

The displacement returned by the cell-based bounding potential might move the active point mass out of its cell. The corresponding candidate event time is, however, always preempted by a cell-boundary event whose candidate event time places the active point mass on the next cell boundary in its direction of motion (the corresponding event handler is introduced in Section 4.2.1). Another remark for the event handler that uses a cell-based bounding potential is related to the excluded relative cells of the underlying cell system. As discussed in Section 4.1.1.6, the cell-based bounding potential does not store derivative bounds for these cases. The in-state should thus not contain point masses whose relative cell is excluded (otherwise the event handler raises an exception). Since the in-state identifiers for the event handlers are constructed in the activator, this element of JF should not generate such problematic in-states for the event handler with the cell-based bounding potential. (For the tag activator of JF-V1.0, this is achieved by the taggers listed in Appendix A.)

The `TwoLeafUnitEventHandlerWithPiecewiseConstantBoundingPotential` dynamically constructs a piecewise constant bounding potential. For the time-sliced position $\mathbf{r}_a$ of the active point mass with velocity $\mathbf{v}$ and the position $\mathbf{r}_t$ of the non-active point mass, the event handler assumes a constant bounding event rate given by

$$q_{M,a}^{\text{Bounding}} = \max\left[q_{M,a}(\mathbf{r}_a, \mathbf{r}_t),\, q_{M,a}(\mathbf{r}_a + \mathbf{v}\Delta t, \mathbf{r}_t)\right] + c. \tag{4.9}$$

Here, $q_{M,a}(\mathbf{r}_a, \mathbf{r}_t)$ is the event rate of the factor potential that should be bounded (which can be easily obtained using the `derivative` method). The constant bounding event rate $q_{M,a}^{\text{Bounding}}$ is straightforwardly integrated to first compute the candidate event time and to afterwards confirm the out-state. The interval length $|\mathbf{v}|\Delta t$ (which is at the same time an upper bound for the maximal displacement of the active point mass) and the constant c are set on initialization of the event handler. These should be chosen in a way that, on the one hand, the bounding event rate certainly provides an upper bound and, on the other hand, the proportion of unconfirmed events remains small.

A non-inverted factor potential (that corresponds to a given factor type) may be associated with several bounding potentials during one run of the application. Consider, e.g., a distance-dependent potential between all pairs of point masses which diverges if the two interacting point masses are at the same position. It is then possible to associate every pair of interacting point masses in non-neighbored cells with a cell-based bounding potential. Every pair in neighbored cells, however, has to be associated with a non-cell-based bounding potential because the cell-based one would diverge. The distinction is carried out in the activator who creates the in-state identifiers of the different event handlers.

4.1.4 Cell-Veto Event Handler

The `LeafUnitCellVetoEventHandler` computes cell-veto events for a pairwise distance-dependent interaction in a system with periodic boundaries. It realizes a set of pair factors with the same factor type that all include the same independent active point mass as discussed in Section 2.6. Like the cell-based bounding potential in Section 4.1.1.6, this event handler relies on a periodic cell system. The event handler iterates similarly over all non-excluded relative cells of the `zero_cell` as well as over all possible directions of motion, and receives derivative bounds from an estimator. The resulting cell-event rates are then used to set up Walker's algorithm using the `Walker` class (once per possible direction of motion). This class provides the total cell-event rate, which is independent of the active point mass's cell for the considered distance-dependent factor potential. It also allows to sample a relative cell with a probability proportional to the corresponding cell-event rate.

On a candidate-event-time request, the event handler only receives the part of the in-state necessary to compute the candidate event time, that is, the branch of the active point mass. Its lifting information and the total cell-event rate for the relevant direction of motion provide, together with an exponentially distributed random number, the candidate event time [see Eq. (2.75)]. Also, Walker's algorithm provides the relative cell of the vetoing factor. Translating the relative cell with respect to the cell of the active point mass using the `translate` method of the underlying cell system yields the target cell, which is returned together with the candidate event time. The mediator determines the target point mass present in the target cell using the activator (if it exists) and its branch accompanies the out-state request. The event handler can then construct the out-state after a confirmation of the event in a similar manner like the event handlers with a bounding potential in Section 4.1.3.

The cell-veto event handler is a good example to justify the seemingly complicated communication with the mediator. It would be possible, in principle, to transmit one in-state per factor in the set of factors treated by the event handler. The cell-veto event handler can then compute the out-state of the vetoing factor without receiving new information in the `send_out_state` method. However, the argument of the `send_event_time` method would be a great part of the full global state, in conjunction with the corresponding need to copy the data of a large portion of the full global state for each candidate-event-time request.

4.1.5 Event Handlers for Factors M with $|I_M| > 2$

So far, the discussed event handlers realized factors or sets of factors that involved only two point masses, that is, (sets of) factors M with an index set $|I_M| = 2$. This led to a trivial out-state computation: For confirmed events, the velocity and the time stamp of the independent active point mass are transferred to the non-active point mass and both branches are kept consistent (which can and should be conveniently done in separate methods so that this is only implemented once). However, event handlers realizing factors M involving more than two point masses, i.e., their index set $|I_M| > 2$, need to compute lifting probabilities [see Eqs. (2.62) and (2.63)].

JF-V1.0 implements an event handler that dynamically constructs a piecewise constant bounding potential for distance-dependent factor potentials between more than two point masses (in the simulations in Section 6.3, this event handler is used for the three-body bending potential of Section 4.1.1.5). This is done in a similar manner like the event handler for pair factor potentials in Section 4.1.3. At first, Eq. (4.9) yields the constant bounding event rate (the different event rates of the factor potential just have more arguments as more point masses are involved). The following computation of the candidate event time and the confirmation of an event remains the same.

The construction of the out-state for a confirmed event, however, demands the computation of the derivative in the direction of motion of the active point mass with respect to all involved point masses. Each derivative is evaluated at the positions of the point masses time-sliced to the event time. (Here, also negative derivatives are important. Hence, the potentials implement a `derivative` method and not a method that returns event rates.) These derivatives, together with the identifier of the point mass with respect to which it was calculated, are then used to fill an instance of the `Lifting` class. Afterwards, this class determines the identifier of the next active point mass according to the correct lifting probabilities. This determines the out-state returned to the mediator.

The `RatioLifting` class uses the lifting probabilities in Eq. (2.65) to determine the identifier of the next active point mass. The absolute values of the negative derivatives and their identifiers are stored in a list. Drawing a uniformly distributed random number between 0 and the sum of all values

in the list then determines the summand in the list, and thus the identifier of the next active point mass. JF-V1.0 further implements the possible choices of the lifting probabilties introduced in [7].

Moreover, event handlers realizing an aforementioned special kind of factor with $|I_M| > 2$ are implemented. One of them uses a cell-based bounding potential; another one computes cell-veto events. In the factor a pairwise factor potential is summed for all pairs of point masses in two different composite point objects. Two of the estimators presented in Section 4.1.1.6 are able to compute cell-based upper bounds on the derivatives for such factors. The two event handlers can thus compute the candidate event time as the similar event handlers for the case of $|I_M| = 2$. The confirmation of the event on an out-state request, however, involves a sum over the derivatives of the pairwise factor potential with respect to the active point mass along its direction of motion. Here, the derivative is evaluated one after another at the separations between the active point mass in one composite point object and all target point masses in the other composite point object, respectively. Similar summations are carried out to finally determine the out-state with the help of an instance of the `Lifting` class.

These event handlers are typically used together with a cell-occupancy system in the activator that tracks composite point objects instead of point masses. The activator just returns the identifiers of composite point objects that are stored within the cell-occupancy system when the in-state identifiers are requested. The tree state handler then constructs branches of these which automatically include all relevant point masses. (For the case of the cell-veto event handler the in-state of a single factor in the set of factors treated is, as explained before, constructed in two steps.)

One can also directly construct a bounding potential for the same kind of "summed" factors. A bounding event rate is given by the sum of the bounding event rates for the pairwise factor potentials between the active point mass located in one composite point object and every point mass located in the other composite point object. This is implemented in a separate event handler of JF-V1.0 that can then realize, for example, such factors when the target composite point object is located in an excluded relative cell with respect to the composite point object that contains the active point mass.

4.1.6 Event Handlers for the Rigid Motion of Composite Point Objects

The event handlers of the previous sections were implemented for the case of a single active point mass with an independent velocity. If the tree state handler furthermore stores composite point objects, their velocity was just induced and the event handlers just had to keep the time-slicing of the in-state branches consistent. As a showcase for the collective-motion possibilities of ECMC integrated into JF by the use of composite point objects, JF-V1.0 implements event handlers that can be used with independent active composite point objects (and thus more than one induced active point masses).

The `RootUnitActiveTwoLeafUnitEventHandler` realizes a single pair factor $M = (\{i, j\}, T_M)$, where i and j are point masses. Only one of them is induced active. (Otherwise all lifting moves must be rejected anyways.) Therefore, i and j belong to different composite point objects, of which only one is independent active. The corresponding distance-dependent factor potential is inverted and this event handler is thus very similar to the corresponding event handler in Section 4.1.2. It receives the same in-state, i.e., the branches of the involved point masses, and computes the candidate event time in the same manner. To construct the out-state, however, the velocity and time stamp of the independent active composite point object including i are transferred to the composite point object including j, while keeping their branches consistent. These branches accompany the out-state request. A similar event handler with the same lifting move between composite point objects in its out-state computation is implemented for the special kind of "summed" factor of the previous Section 4.1.5.

4.2 Event Handlers for Pseudo-Factors

Introducing pseudo-factors into ECMC allows to formulate its time evolution entirely in terms of events. Still, there is a crucial distinction between event handlers realizing a single pseudo-factor or a set of these. For the former, the full in-state is known at the candidate-event-time request, while for the latter, it is only known at the out-state request.

4.2.1 Cell-Boundary Event Handler

As discussed in Section 3.5.1, cell-occupancy systems in JF-V1.0 stored in the activator have to be kept strictly consistent with the global state stored in the state handler. This is enforced by cell-boundary events of the `CellBoundaryEventHandler` (see Fig. 3.2). In order to compute its events, the event handler has access to the cell system and the tracking level of the cell-occupancy system. The latter is required because the cell-occupancy system can track both point masses and composite point objects. Several cell-occupancy systems require several cell-boundary event handlers initialized with different cell systems and tracking levels.

The call of the `send_event_time` method is accompanied by a single branch with a single active unit on the tracking level of the cell-occupancy system. (This active unit, which either refers to a point mass or a composite point object, can have both an independent or an induced velocity.) Hence, a cell-boundary event handler realizes a single pseudo-factor depending on a single identifier. The candidate event time corresponds to the time-sliced minimal position of the active unit that belongs to the successor cell in its direction of motion. (For this, the cell system's `successor` and `cell_min` methods are used.) This also builds the out-state.

After the out-state of a cell-boundary event has been committed to the global state, the following update of the cell-occupancy system in the activator will place the tracked active unit in the new cell and thus keeps the internal state consistent.

4.2.2 Event Handlers connected to Output Handlers

Output handlers in the input-output handler should be run after the out-state of events of certain event handlers have been committed to the global state (step 9 in Fig. 3.3). This behavior is achieved by the mediating methods in the mediator which only get executed depending on the class of event handlers whose out-state was committed (see Section 3.7).

The mediator implements such a mediating method for the abstract `SamplingEventHandler` class. This class is furthermore connected to an output handler, i.e., it possesses a property with the name of an output handler. After an event from an (inheriting) event handler, the corresponding mediating method passes the full global state from the state handler to the output handler with the given name (via the input-output handler). This output handler can then use the global state to sample an observable (see Section 3.6.2).

The `FixedIntervalSamplingEventHandler` of JF-V1.0 creates sampling events with equally spaced candidate event times. The spacing, or rather the sampling duration, is set on initialization. The implementation of the `send_event_time` method does not rely on any in-state because it just uses the candidate event time that was returned last. The out-state request is accompanied by the branches of all independent active point masses or composite point objects returned by the `extract_active_global_state` method of the tree state handler. (Since both point masses and

composite point objects may be independent active for this event handler, both cases are just referred to by independent active units in the following.) In the out-state, all units that appear in the transmitted branches are time-sliced to the event time.

The sampling event handler of JF-V1.0 can be viewed as realizing a set of pseudo-factors which depends on the identifiers of all *possible* independent active units. Only at the time of the out-state request, the actually independent active units are known and can be transmitted. Another (more costly) implementation could implement the `send_event_time` method with an in-state given by the branches of all independent active units. These would be stored for the following out-state request so that the `send_out_state` method does not require any arguments. This would necessitate, however, the recomputation of the candidate event time whenever an independent active unit changes.

The mediator defines a different mediating method for the abstract `DumpingEventHandler` class that allows to store the state of an entire run of the application in an output handler (see Section 3.6.2). The `FixedIntervalDumpingEventHandler` creates corresponding events in a fixed interval in the same manner as the previously discussed sampling event handler.

4.2.3 End-of-Chain Event Handler

An end-of-chain event handler stops an event chain and initializes a new one after a given chain duration. The next candidate event time can be returned without any in-state and just depends on some distribution of chain durations. The event handler also samples the identifier of the desired independent active unit at the beginning of the new event chain. This identifier is returned together with the candidate event time. (The event handler is therefore aware of all possible identifiers of the state handler, see Section 5.1).

The branch of this unit is received as the argument of the `send_out_state` method together with the branch of the currently independent active unit. To create the out-state, the event handler samples a new direction of motion based on the old direction to construct a new velocity with the same absolute value as before. All units in the branch of the current independent active unit are time-sliced to the event time and their velocity is set to zero. Then, the new velocity and time stamp of the beginning independent active unit are set and its branch is kept consistent. Such an event handler realizes again a set of pseudo-factors (see Fig. 4.1). JF-V1.0 implements the abstract `EndOfChainEventHandler` class that implements the described actions. An inheriting class just has to define methods that determine the new chain duration, the identifier of the new independent active unit, and the new direction of motion, respectively. The `SameActivePeriodicDirectionEndOfChainEventHandler` of JF-V1.0 maintains the current independent active unit and just changes its direction of motion in a periodic manner. Hence, this event handler does not return anything besides the candidate event time in the `send_event_time` method.

4.2.4 Start/End-of-Run Event Handlers

The only obligatory event handler in a run which uses the tag activator of JF-V1.0 (see Section 3.5) is an event handler inheriting from the abstract `StartOfRunEventHandler` class. Its out-state is the first one committed to the global state. Also, the tag activator uses this event handler as an entry point to determine which events should be activated and created thereafter.

JF-V1.0 implements the `InitialChainStartOfRunEventHandler`, which also initializes the global lifting state (the input handlers in Section 3.6.1 only initialized the global physical state). On

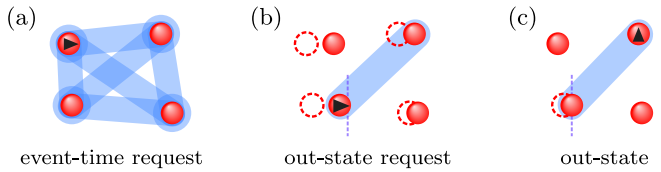


Figure 4.1: The end-of-chain event handler realizes a set of pseudo-factors that couples the final active unit of the old chain to the beginning active unit of the new chain. (a): At the time of the event-time request, the final active unit of the old chain is not yet known and thus a set of pair pseudo-factors is treated. (b): At the time of the out-state request, the final active unit of the old chain is known and the beginning active unit of the new chain was sampled. Hence, the vetoing pseudo-factor is determined. (c): The new event chain has a different active unit with a new direction of motion. The old active unit is time-sliced to the event time. This figure has been published in [39] (see publication in Appendix B).

initialization, the event handler class receives an identifier of a point mass or composite point object which should be independent active at the beginning of the run. Also, the absolute value of the desired velocity and a direction of motion is set. The `send_event_time` method returns the candidate event time 0.0 and the identifier. The out-state request is accompanied by the branch of the corresponding identifier and the event handler can set the velocity and time stamp to construct the out-state.

An end-of-run event handler inherits from the abstract `EndOfRunEventHandler` class, for which a mediating method is defined in the mediator (step 9 in Fig. 3.3). This mediating method raises an `EndOfRun` exception which terminates the main iteration loop of ECMC in the mediator (see Section 3.7). Before the run is terminated, an observable can be sampled. For this, the end-of-run event handler is connected to an output handler like the event handlers described in Section 4.2.2. The `FinalTimeEndOfRunEventHandler` of JF-V1.0 finishes the run after a fixed final time, which is returned as the candidate event time. An out-state request is accompanied by the branches of all independent active units which are time-sliced to create the out-state.

Events of event handlers that compute a candidate event time without any in-state should only be trashed in the scheduler after their out-state was committed to the global state. Also, the next event of the same event handler should be computed and inserted into the scheduler directly afterwards. Using the tag activator of JF-V1.0, this is easily achieved by accordingly chosen create and trash tag lists. Only the sampling/dumping, end-of-chain, and end-of-run event handlers are affected by this.

4.2.5 Mode Switching Event Handlers

Section 4.1.6 introduced event handlers suited for the rigid motion of entire composite point objects. This implies that the point masses belonging to independent active composite point objects are moved collectively, which renders the algorithm non-ergodic. Mode switching to the case of independent active point masses becomes a necessity in order to have all factors be considered during one run. The mode switching in both directions is achieved by different instances of the `RootLeafUnitActiveSwitcher` of JF-V1.0 (the direction is set on initialization). Hence, during one run of the application two of these are used but the event of only one of them is stored in the scheduler at any time. The `send_event_time` method samples a new candidate event time based on the time stamp of the independent active unit in the in-state. An out-state request is accompanied by the entire tree of the current independent active unit which allows to change the mode in the out-state.

CHAPTER 5

Run Specifications

The last two chapters extensively introduced the various elements of JF and their implementation in JF-V1.0. This chapter summarizes the the last two parts of the application which were only hinted before. First, a limited amount of the elements requires access to globally shared information that is stored in separate Python modules. Those are treated in Section 5.1. By this, for example, the thermodynamic β for a simulation is only set once, and not repeatedly during the initialization of event handlers. (Event handlers realizing factors use β in order to sample the exponentially distributed positive factor potential change ΔU_M^+). Section 5.2 introduces the user interface for each run of the JF application. In JF-V1.0, it consists of configuration files in the INI-file format. This allows to specify physical and algorithmic parameters without the necessity to write Python code.

5.1 Globally Shared Information

Globally shared information of JF-V1.0 is stored in the `setting` package and in specialized modules therein. The former stores information available in every run, whereas the latter specifies different system shapes in different modules (e.g., the parameters of a hypercuboid or hypercubic simulation box). By this, the parameters of a NVT (canonical) physical ensemble are stored, whereby the parameters remain unchanged during a run. Here, N is the number of point masses, V the volume of the simulation box, and T is the temperature. During one run of the application, typically only one of the specialized modules is initialized besides the `setting` package.

The `setting` package stores the thermodynamic β and the dimension of the current simulation. (The latter is used, for example, in the merged-image Coulomb potential of Section 4.1.1.4 that was only implemented for a cubic box in three dimensions. An exception is thrown for inappropriate dimensions.) Furthermore, it stores variables that allow all elements of JF to autonomously construct all possible global state identifiers. This is required, e.g., by the end-of-chain event handler of Section 4.2.3. In JF-V1.0, the stored variables within the `setting` package are specific to the tree state handler and only allow for an arbitrary number of identical trees representing composite point objects of height at most two. This is in accordance with the current restriction of the cell-occupancy systems (see Section 3.5.1), and the currently implemented input handlers (see Section 3.6.1). (Moreover, one of the taggers in Appendix A uses this in order to extrapolate given factors between two composite point objects to factors between *all* of them.) Future versions are going to replace the storing variables by flexible objects that depend on the used state handler.

Several specific system shapes are implemented in as many modules within the `setting` package. JF-V1.0 implements hypercubic and hypercuboid system-shape modules. The former adds a single variable that stores the unique system length. The latter, in contrast, stores a list of system lengths. Both modules moreover implement a method to sample a random position and methods that implement periodic boundaries (these methods correct, for example, a position for periodic boundaries or compute the minimum separation vector between two positions within the system box).

Each system-shape module defines a "setter" class that initializes both the module itself and the `setting` package properly in its `__init__` method. Arguments of this method are β, the dimension, and the variables specific to the system shape. The identifier variables, however, are initialized by the input handlers. (That is because if an input handler, e.g., reads an initial global physical state from a file, the according parameters are already encoded there.) During one run of the application only a single setter class is used. This allows to use elements of JF within the run whose implementation uses variables from the `setting` package or the chosen system-shape module.

The elements of JF that just require valid random positions (like the input handlers in Section 3.6.1 that create a random initial global physical state) or just rely on periodic boundaries (like the event handlers in Chapter 4) should be, nevertheless, independent of the system-shape modules. For the case of the periodic boundaries, this is achieved by defining the abstract `PeriodicBoundaries` class that only possesses abstract methods. All system-shape modules then implement an inheriting class thereof. The setter class of a specific system-shape module sets an instance of this inheriting class directly in the `setting` package. Similarly, the method creating random positions is broadcast.

By this, there is a very limited amount of elements that can only be used if the correct system shape was chosen (that is, only a limited amount of modules, within which the elements are implemented, import the specific system-shape modules). The merged-image Coulomb potential and its bounding potential are only implemented, as mentioned, for a cubic simulation box. The cuboid periodic cells (see Section 3.5.1) use the hypercuboid system-shape module. (It is noteworthy, however, that choosing the hypercubic simulation box also initializes the hypercuboid module so that the cuboid periodic cells are usable for both system shapes.) Since a pdb file can only specify hypercuboid system shapes, the corresponding output handler explicitly uses the hypercuboid module as well (see Section 3.6.2).

Note that the current `setting` package and the system-shape modules do not allow for changes of the stored NVT physical parameters during one run of the application. Moreover, it is not possible to store other physical parameters (like, e.g., the pressure P). This has to be considered in future versions of the JF application.

To end this section (and to properly discuss all parts of the JF application), the `base` package is mentioned. Within this package, modules with several helper classes and functions are located. This includes, for example, functions acting on vectors. However, the `base` package does not store any additional data which distinguishes it from the `setting` package.

5.2 Configuration Files

The executable `run.py` script of JF starts a run of the application. The first positional argument of this script is a configuration file in the INI-file format that specifies the physical and algorithmic parameters of the run (i.e., the temperature, system shape and size, dimension, type and number of composite point objects and point masses, event handlers, factor potentials, lifting schemes, total run time, sampling frequency, etc.).

(a)

```
class SingleProcessMediator:
      def __init__(self, input_output_handler:  InputOutputHandler, state_handler:  StateHandler,
                         scheduler:  Scheduler, activator:  Activator):
```

(b)

```
[section] property value
```

```
[Run]
mediator = single_process_mediator
setting = hypercubic_setting

[HypercubicSetting]
system_length = 1
beta = 2
dimension = 3

[SingleProcessMediator]
state_handler = tree_state_handler
scheduler = heap_scheduler
activator = tag_activator
input_output_handler = input_output_handler

[TagActivator]
taggers =
        coulomb (factor_type_map_in_state_tagger),
        sampling (no_in_state_tagger),
        end_of_chain (no_in_state_tagger),
        start_of_run (no_in_state_tagger),
        end_of_run (no_in_state_tagger)

[Coulomb]
event_handler = two_leaf_unit_bounding_potential
                _event_handler
```

```
[TwoLeafUnitBoundingPotentialEventHandler]
potential = merged_image_coulomb_potential
bounding_potential = inverse_power_coulomb
                     _bounding_potential

[Sampling]
event_handler = fixed_interval_sampling_event_handler

[FixedIntervalSamplingEventHandler]
sampling_interval = 0.56789
output_handler = separation_output_handler

[EndOfChain]
event_handler = same_active_periodic_direction
                _end_of_chain_event_handler

[SameActivePeriodicDirectionEndOfChainEventHandler]
chain_length = 0.78965

[TreeStateHandler]
physical_state = tree_physical_state
lifting_state = tree_lifting_state

[InputOutputHandler]
output_handlers = separation_output_handler
input_handler = random_input_handler
```

Figure 5.1: An exemplary configuration file in the INI-file format that specifies the physical and algorithmic parameters of a run of the JF application. (This figure shows excerpts of a configuration file that is part of JF-V1.0: `src/config_files/2018_JCP_149_064113/coulomb_atoms/power_bounded.ini`. This file starts a simulation treated in Section 6.1.1.) (a): A typical `__init__` method of a JF class including type hints. (b): Excerpts of the configuration file where some lines are split for clarity. The obligatory `[Run]` section specifies the mediator and the setting. The ensuing sections correspond to classes of JF. Within a section, a property specifies the argument name in the class's `__init__` method and the corresponding value the argument itself. This figure has been published in [39] (see publication in Appendix B).

Within the architecture of JF-V1.0, all parameters are set during the initialization of the different elements, i.e., as arguments of the `__init__` methods of the respective classes (e.g., the temperature is set on initialization of the setting class, and the end-of-run time is set on initialization of the end-of-run event handler). A configuration file is thus composed of sections that each corresponds to a class. A section contains pairs of properties and values. The former corresponds to the name of the argument in the respective class's `__init__` method, the latter provides the argument itself (see Fig. 5.1).

The content of the configuration file is parsed by the Python `configparser` module and passed to the factory of JF (which is located in the `factory` module in the aforementioned `base` package). The values used as the arguments in the initialization methods are parsed as strings and have to be converted by the factory. To do so, it inspects the (mandatory) type hints in the respective `__init__` method. The factory of JF-V1.0 allows for the most usual basic data types (i.e., `bool`, `float`, `int`, `str`) as well as lists of these (given as a comma separated list within the configuration file).

Furthermore, the values can refer to classes that are part of JF (or lists thereof). The constructed argument by the factory is then an instance of the corresponding class. If the initialization of this instance requires arguments itself (because the corresponding `__init__` method expects arguments), these are set in a similarly named section of the same configuration file.[1] This implies a tree representation of the sections that appear in the configuration file of a certain run (see Fig. 5.2).

[1] According to PEP 8, class names (and hence, section names) are given in CamelCase. In contrast, argument names of the `__init__` method (and hence, properties and values within the configuration file) are given in snake_case.

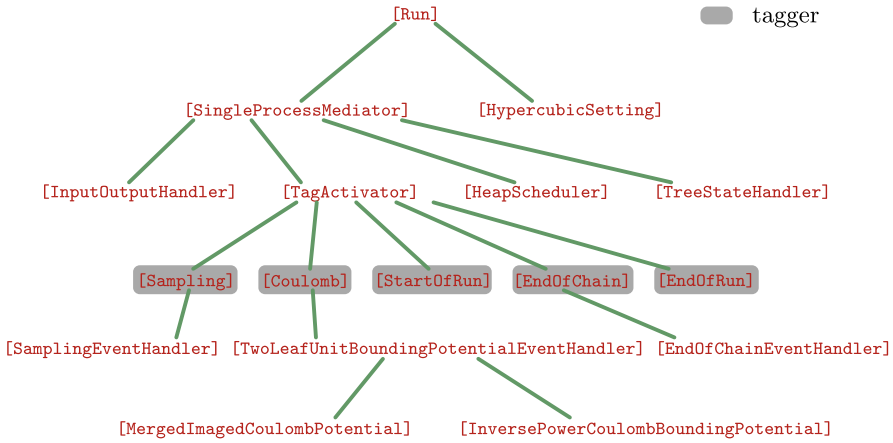


Figure 5.2: Tree representation of the configuration file shown in Fig. 5.1. Only a part of the tree is shown and some event handler names are shortened. The children of the `[TagActivator]` section correspond to all declared taggers of a run, which are itself associated to event handlers (see Section 3.5). Their tags are given by the section names and can thus appear in the create and trash tag lists of all taggers. This figure has been published in [39] (see publication in Appendix B).

If the same class should be used with different initialization arguments during a run (this usually occurs for classes of taggers that should be associated with different event handlers), the factory allows for aliases. These are then used for both the value and the section names. The name of the class that should be initialized by the factory follows the value in parentheses. The factory moreover changes the name of the class to the alias during the run. (This is very convenient for the taggers of the tag activator, introduced in Section 3.5, that use the class name as its tag, if not specified otherwise.)

In the configuration file, the only obligatory `[Run]` section specifies the mediator (i.e., either the `SingleProcessMediator` or the `MultiProcessMediator`) and the setter class of the system-shape module. The ensuing sections then choose the parameters in the `__init__` methods of the two corresponding classes. The order of the sections is irrelevant.

The application typically creates output in files using the output handlers (see Section 3.6.2). Run-time information is logged by using the Python `logging` module either to standard output or to a file, and takes place on a variety of levels from `DEBUG` to `INFO` to `WARNING`. Both the output and the logging level can be specified using optional command-line arguments of the `run.py` script. Logging information ranges from the identification of the used CPUs to the initialization information of classes to run-time information within the main iteration loop of ECMC (e.g., the event handler that created the event with the shortest candidate event time in each iteration). Also, a unique identification hash is part of the logging output (created by using the Python `uuid` module). This hash is furthermore a part of all created output files which allows to uniquely link them to log files.

Besides the `run.py` script, there also exists the executable `resume.py` script in JF-V1.0. This allows to restart a run given the file created by the dumping output handler of Section 3.6.2 as the first positional argument. (Naturally, this script likewise relies on the Python dill package [78, 79].) It is noteworthy that the `resume.py` script can choose a logging level independently and does therefore not necessarily have the same level as the run which created the used file. By this, the `resume.py` script with a chosen higher logging level is predestined for debugging.

CHAPTER 6

All-Atom Simulations Using the Application

This chapter showcases the flexibility and potential of the JF application already in its first version. To do so, a variety of all-atom models are simulated in a multitude of ways. Section 6.1 treats simulations for two electrically charged point masses that interact via the long-ranged Coulomb interaction. Section 6.2 considers simulations of two finite-size dipoles, followed by simulations for two water molecules in Section 6.3. The results reproduce published data [7].

The used configuration files to produce the data presented in this chapter are part of JF-V1.0 and thus publicly available within the repository on GitHub. For this reason (and since some of these are rather lengthy) they are not a part of this thesis. Instead, their implemented logic is explained in detail.

All configuration files use the single-process mediator described in Section 3.7. However, the configuration files are easily modified to use the multi-process version (see Section 5.2). Furthermore, all files choose the hypercubic system-shape module in three dimensions. The choices of the heap scheduler (see Section 3.4), tree state handler (see Section 3.2) and tag activator (see Section 3.5) are self-explanatory as they are the sole implementations of the corresponding elements in JF-V1.0. The initial global physical state is created randomly by the corresponding input handler (see Section 3.6.1).

Differences in the configuration files occur for the chosen event handlers which realize different factors and pseudo-factors. As discussed in Section 3.5, the used tag activator associates each event handler with a tagger that, on the one hand, generates its in-state identifiers (which are transformed to in-states using the tree state handler). On the other hand, taggers associate event handlers and their events with a tag. These tags are used in the create and trash tag lists of all taggers. By this, it is determined which events are trashed in the scheduler after an event of a certain event handler was committed to the global state, and which events have to be computed in the corresponding event handlers in the following leg of ECMC's piecewise non-interacting time evolution. Note that events and event handlers are referred to by their associated tags in this chapter.

All configuration files have unique start-of-run, end-of-run, end-of-chain, and sampling event handlers that all realize pseudo-factors (see Section 4.2). Each of them is associated with a tagger that does not generate any in-state identifier and associates an accordingly chosen tag (e.g., the tag of the start-of-run event handler is `start_of_run`). After the initial start-of-run event, event handlers realizing all factors and pseudo-factors have to compute events. In contrast, only the just committed event is trashed in the scheduler. After a sampling event, solely the same event is trashed and recomputed. An end-of-chain event is more complex because it changes the velocity of the currently independent active point mass or composite point object. Therefore, all factor events need

to be trashed in the scheduler and to be recomputed in the corresponding event handlers afterwards (together with a new end-of-chain event). Since an end-of-run event finishes a run of the application, it is irrelevant which events should be trashed and computed afterwards. The described create and trash decisions are valid for all simulations in this chapter and are encoded in the tag lists of the corresponding taggers.

The data analysis of the observables in each section of this chapter follows a simple and robust method. Instead of using, e.g., the blocking method of Section 2.1.3 to estimate the error of a sample average of a single run, each presented simulation was run 24 times. The *uncorrelated* sample averages of each run can then be treated with Eq. (2.24). (Naturally, the sample averages are calculated after the Markov chain initially reached the equilibrium distribution.) Additionally, a reversible Markov-chain simulation (using the Metropolis algorithm) was run for every model. Here, the error was estimated using the aforementioned blocking method. This simulation was run for a relatively long simulation time in order to obtain a result with a relatively small standard error. This result is then used to verify the results of the ECMC simulations (where the simulation times were balanced between being reasonable long and having a comparable standard error as the reversible simulation).

All of the simulations in this chapter use the merged-image Coulomb potential of Section 4.1.1.4. In order to ensure that the chosen cut-offs of the sums in real and Fourier space do not introduce a systematic error, the ECMC simulations were also repeated with larger cut-offs.

6.1 Interacting Charged Point Masses

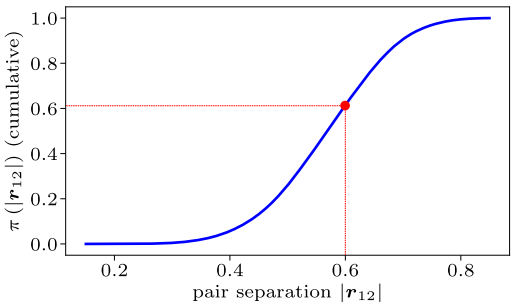
This section treats the ECMC sampling of the Boltzmann distribution for two identical charged point masses that interact via the merged-image Coulomb pair potential of Section 4.1.1.4 and are described by a single Coulomb pair factor.[1] The side length of the three-dimensional cubic simulation box is chosen as $L = 1$. In addition to that, periodic boundary conditions are used and $\beta c_1 c_2 = 2$, where c_1 and c_2 are the electric charges of the two point masses. At each time, one of the two point masses is active and moves parallel to one of the Cartesian coordinate axes (the initially active point mass and its velocity is chosen by the start-of-run event handler). All trees are trivial in the tree state handler as they contain only a single node. This also applies to the constructed in-state branches, which then contain just a single cnode.

Statistically equivalent output is obtained by associating the non-invertible merged-image Coulomb potential with its bounding potential in Section 6.1.1, or else with a cell-based bounding potential in Section 6.1.2. This will require the usage of a cell-occupancy system. Section 6.1.3 then uses the cell-veto algorithm for the treated pair factor. In all three simulations, a sampling event is followed by computing a sample of the minimum separation between the two point masses in an output handler (see Fig. 6.1 for a comparison of the results of the different simulations).

6.1.1 Merged-Image Coulomb Bounding Potential

This most simple `power_bounded.ini` configuration file requires only a single factor event handler that has access to both the merged-image Coulomb and its bounding potential in order to compute events (see the discussion of the `TwoLeafUnitBoundingPotentialEventHandler` in Section 4.1.3). The

[1] The used configuration files are located in the `src/config_files/2018_JCP_149_064113/coulomb_atoms` directory of JF-V1.0.



method	$\pi(\lvert\mathbf{r}_{12}\rvert < 0.6)$
①	0.61305 ± 0.00003
②	0.61306 ± 0.00006
③	0.61299 ± 0.00008
④	0.61307 ± 0.00012

Figure 6.1: Cumulative histogram $\pi(|\mathbf{r}_{12}|)$ of the minimum pair separation $|\mathbf{r}_{12}| = |\mathbf{r}_2 - \mathbf{r}_1|$ between the nearest images of two electrically charged point masses 1 and 2 in a three-dimensional cubic simulation box with periodic boundary conditions. The results reproduce published data [7, Fig. 8]. Moreover, $\pi(|\mathbf{r}_{12}| < 0.6)$ including standard errors is shown for each simulation method. The obtained values agree well with the weighted average of all values within their respective standard errors. ①: Reversible Markov-chain Monte Carlo. ②: Method of Section 6.1.1. ③: Method of Section 6.1.2. ④: Method of Section 6.1.3. This figure has been published in [39] (see publication in Appendix B).

corresponding tagger associates the `coulomb` tag and generates the identifiers of the two point masses to construct the in-state. After the out-state of a `coulomb` event has been committed to the global state, only the very same event is trashed in the scheduler. Also, only the `coulomb` event handler computes a new event afterwards. (Of course, it would be feasible to hard-wire the determination of which event handler needs to compute an event and which events should be trashed for this simple system. Nevertheless, the tag activator is used to give an easy introduction.)

The configuration file can be modified to simulate N point masses with a Coulomb pair factor for each pair of point masses. The input handler has to create a corresponding global physical state initially, and the `coulomb` tagger requires $N - 1$ instances of its event handler. When the in-state identifiers are requested, the tagger generates all identifier pairs that include the currently active point mass.

6.1.2 Cell-Based Bounding Potential

The `cell_bounded.ini` configuration file uses a cell-occupancy system within the tag activator (see Section 3.5.1). The size of the underlying cell system is arbitrarily chosen for this showcase. For each cell, the directly neighbored cells are excluded due to the singularity of the merged-image Coulomb potential at vanishing separations of the point masses. The cell-occupancy system tracks all point masses independently of their charges.

The Coulomb pair factor is realized by two different event handlers during one run of the application. This changes based on the active and the target cell (that is, the cell of the active and non-active point mass, respectively) at the moment of the call of the activator's `get_event_handlers_to_run` method. If the target cell is not excluded with respect to the active cell, the `coulomb_cell_bounding` tagger generates the two identifiers to construct the in-state for its event handler. This event handler (the `TwoLeafUnitCellBoundingPotentialEventHandler` of Section 4.1.3) uses a cell-based bounding potential besides the merged-image Coulomb potential in order to compute events. The cell-based derivative bounds are computed at the beginning of the run using the boundary-point estimator of Section 4.1.1.6.

Otherwise the target cell is excluded with respect to the active cell. Then, the `coulomb_nearby` tagger generates the in-state identifiers for its event handler that computes an event in the same way as the event handler of the previous Section 6.1.1.

Modifying this configuration file to $N > 2$ interacting point masses possibly introduces surplus point masses in the internal state. The `coulomb_surplus` tagger associates the corresponding Coulomb pair factors with the merged-image bounding potential (although it would also be possible to use a cell-based bounding potential). The corresponding section in the configuration file that initializes this tagger is already included in the $N = 2$ version available in JF-V1.0.

The `CellBoundaryEventHandler` of Section 4.2.1 computes cell-boundary events to keep the internal state consistent with the global state. The associated `cell_boundary` tagger just generates the identifier of the currently active point mass to construct the in-state. After `cell_boundary`, `coulomb_cell_bounding`, `coulomb_nearby`, and `coulomb_surplus` events, events with all of these tags are trashed and recomputed (if the corresponding taggers generate in-state identifiers).

6.1.3 Cell-Veto Algorithm

The `cell_veto.ini` configuration file uses the cell-veto algorithm to treat a set of Coulomb pair factors at once. The set contains every pair factor between the active point mass in its active cell and hypothetical target point masses in non-excluded cells of the active cell. This simulation uses a similar cell-occupancy system as in the previous Section 6.1.2. Moreover, the `LeafUnitCellVetoEventHandler` of Section 4.1.4 uses the same boundary-point estimator.

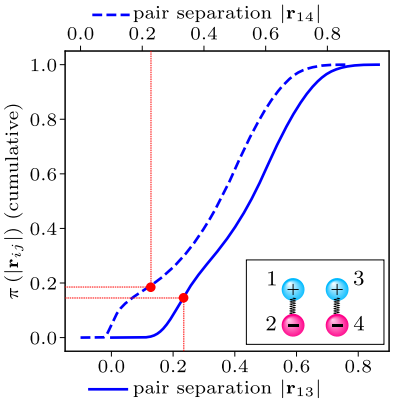
The cell-veto event handler computes the candidate event time of the cell-veto event based on the cnode branch of the active point mass. The corresponding in-state identifier is generated by the associated `cell_veto` tagger. Together with the computation of the candidate event time, the target cell is sampled. Since there is only a single Coulomb pair factor, this target cell does typically not contain the non-active point mass and the corresponding cell-veto event is very often rejected. (This leads, together with the fact that a new iteration in the mediator is started also after unconfirmed events, to a slowed down simulation. This increases its standard error in Fig. 6.1 because the simulation times were nevertheless chosen similarly to the other methods of this section.)

The remaining part of this configuration file is very similar to the one of Section 6.1.2, including the `coulomb_nearby`, `coulomb_surplus`, and `cell_boundary` taggers and their tag lists. The `coulomb_surplus` tagger is again only relevant if the configuration is modified to $N > 2$.

6.2 Interacting Dipoles

The configuration files of this section sample the Boltzmann distribution for two identical finite-size dipoles in a three-dimensional cubic simulation box with side length $L = 1$ and periodic boundary conditions.[2] The value of the thermodynamic beta is chosen as $\beta = 1$. In the considered model, charged point masses in different dipoles interact via the merged-image Coulomb potential (pairs 1–3, 1–4, 2–3, and 2–4 in Fig. 6.2). Here, the charges of the point masses in a dipole are chosen as $c_i = \pm 1$. The point masses within each dipole interact via the displaced-even-power-law potential of Section 4.1.1.3 with the parameters $p_{\mathrm{DEPP}} = 2$, $k_{\mathrm{DEPP}} = 200$, and $r_0 = 0.1$ (pairs 1–2 and 3–4).

[2] The used configuration files are located in the `src/config_files/2018_JCP_149_064113/dipoles` directory of JF-V1.0.



method	$\pi(\lvert\mathbf{r}_{13}\rvert < 0.22)$	$\pi(\lvert\mathbf{r}_{14}\rvert < 0.22)$
①	0.12128 ± 0.00003	0.18453 ± 0.00003
②	0.12129 ± 0.00003	0.18458 ± 0.00004
③	0.12130 ± 0.00002	0.18457 ± 0.00003
④	0.12129 ± 0.00003	0.18456 ± 0.00005
⑤	0.12108 ± 0.00037	0.18412 ± 0.00053

Figure 6.2: Cumulative histograms $\pi(|\mathbf{r}_{13}|)$ and $\pi(|\mathbf{r}_{14}|)$ of the minimum pair separations $|\mathbf{r}_{ij}| = |\mathbf{r}_j - \mathbf{r}_i|$ between the nearest images of two electrically charged point masses i and j in different finite-size dipoles (see inset). The simulations used a three-dimensional cubic simulation box with periodic boundary conditions. The results reproduce published data [7, Fig. 11]. Moreover, $\pi(|\mathbf{r}_{13}| < 0.22)$ and $\pi(|\mathbf{r}_{14}| < 0.22)$ including standard errors are shown for each simulation method. The obtained values agree well with the corresponding weighted average within their respective standard errors. ①: Reversible Markov-chain Monte Carlo. ②: Method of Section 6.2.1. ③: Method of Section 6.2.2. ④: Method of Section 6.2.3. ⑤: Method of Section 6.2.4. This figure has been published in [39] (see publication in Appendix B).

This potential yields a fluctuation of the sampled dipole separation around r_0 (this potential is called harmonic in the following). A repulsive short-range potential (using the inverse-power-law potential of Section 4.1.1.1 with the parameters $k_{\text{Inverse}} = 10^{-6}$, $p_{\text{Inverse}} = 6$, and without charges) between oppositely charged point masses in different dipoles counterbalances the attractive Coulomb potential at small distances (pairs 1–4 and 2–3). For the two latter potentials only the closest images between the involved point masses are considered. That is because the repulsive short-range potential decays quickly with distance whereas the intra-dipole harmonic potential should only act between point masses that together build a dipole (and not between point masses in different periodic images of the dipole). Each dipole is represented in the tree state handler by a composite point object made up of two oppositely charged point masses.

Statistically equivalent output for this model is obtained in the following ways: Section 6.2.1 considers pair factors for all interactions. Section 6.2.2 combines the Coulomb factors to a single dipole-dipole Coulomb factor and associates it with a cell-based bounding potential. The same type of factor is treated by the cell-veto algorithm in Section 6.2.3. All of these three sections use a single active point mass at any time. The final Section 6.2.4, in contrast, alternates between concurrent moves of the entire dipoles with moves of the individual point masses. This showcases the collective-motion possibilities of ECMC that are integrated into the JF application. All of the simulations sample the minimum separation between the equally and oppositely charged point masses in the two dipoles (see Fig. 6.2 for a comparison of the results of the different simulations).

6.2.1 Coulomb Pair Factors

Like in the simulations for two interacting charged point masses in Section 6.1, during a run of the `atom_factors.ini` configuration file one of the four point masses within the two finite-size dipoles is active at any time. This point mass has a velocity parallel to one of the Cartesian coordinate axes.

The pair factors corresponding to the inverted harmonic and repulsive factor potentials are realized in separate instances of the `TwoLeafUnitEventHandler` of Section 4.1.2 (tagged `harmonic` and `repulsive`, respectively). This event handler computes the minimum separation between the pair of point masses as desired by the underlying dipole model. Based on the currently active point mass, their taggers determine the in-state identifiers for the event handlers. The constructed in-states then consist of branches of the two involved point masses (including the cnodes of the dipole barycenters). Since there is only a single pair factor of both types that involves the currently active point mass, a single event handler instance is required in each tagger. The two Coulomb pair factors involving the active point mass are realized one by one as in Section 6.1.1. For this, the `coulomb` tagger requires two instances of the corresponding event handler.

The create and trash tag lists of all the discussed taggers are straightforward because after a factor event (that is, events with the `coulomb`, `harmonic`, or `repulsive` tags) all factor events in the scheduler are trashed. Similarly, all corresponding taggers generate new in-state identifiers for their event handlers, which include the (possibly) new active point mass's identifier.

It is noteworthy that the event handlers keep the in-state branches and thus, the dipole barycenters consistent with the point masses' positions. However, the properties of the composite point objects are never used during the simulation. It would be alternatively possible to simulate four point masses interacting via the different pair factors. (This can, of course, be achieved using JF-V1.0).

6.2.2 Coulomb Dipole Factors, Cell-Based Bounding Potential

The `cell_bounded.ini` configuration file combines the four Coulomb pair factors to a single Coulomb four-body factor. (The factor potential is given by the sum of the merged-image Coulomb potentials between the pairs 1–3, 1–4, 2–3, and 2–4 in Fig. 6.2.) As discussed in Section 4.1.1.6, the factor event rate decays much faster with distance compared to the Coulomb pair factors. Also, the chosen lifting scheme considerably influences the dynamics (see [7, Section IV]). The same Section 4.1.1.6 introduced estimators that determine cell-based bounds on the factor derivative of such a "summed" factor. This configuration file uses the Monte-Carlo estimator. (Note that the derivative bounds of this estimator are not perfect. Very rarely it occurs that the bounding event rate is smaller than the factor event rate during the confirmation of an event. Although this introduces a systematic error, the rarity of these cases renders this neglectable.)

As in the previous Section 6.2.1, there is only a single active point mass at all times. Also, the same `harmonic` and `repulsive` tagged event handlers realize the pair factors with the correspondingly named factor potentials. However, this configuration additionally sets up a cell-occupancy system that tracks the dipole barycenters. This is achieved by initializing its `cell_level` property to 1 (the level 2 would correspond to tracking the point masses). The size of the underlying cell system is again arbitrarily chosen and only directly neighbored cells are excluded.

The `coulomb_cell_bounding` and `coulomb_nearby` taggers use the cell-occupancy system to differentiate between the pair of dipoles being in non-neighbored or neighbored cells. The former case is treated by an event handler using a cell-based bounding potential; the latter case by an event handler that directly constructs a bounding potential using the bounding event rates of the Coulomb pair factors (see Section 4.1.5). Both event handlers are initialized with a lifting scheme that is used to choose the next active point mass. A `coulomb_surplus` tagger generates in-state identifiers including surplus dipole identifiers, which only appear if the configuration file is modified to $N > 2$ dipoles. The corresponding event handler also uses the directly constructed bounding potential.

Similar to the simulation of two charged point masses using a cell-based bounding potential of Section 6.1.2, a `CellBoundaryEventHandler` computes cell-boundary events. However, these events are now based on the time when the dipole barycenter containing the active point mass crosses the next cell boundary in its induced direction of motion. To compute the event, the event handler acquires knowledge about the `cell_level` property. After such a `cell_boundary` event, all events relying on the cell-occupancy system are trashed and recomputed (i.e., events with the tags `cell_boundary`, `coulomb_cell_bounding`, `coulomb_nearby`, and `coulomb_surplus`). The `harmonic` and `repulsive` events, however, should not be trashed and recomputed. In contrast, after any factor event all of the factor events and also the `cell_boundary` event are trashed and recomputed (because the induced active dipole may have changed).

6.2.3 Coulomb Dipole Factors, Cell-Veto Algorithm

The `cell_veto.ini` configuration file uses the same factors and pseudo-factors, as well as the same internal state as the configuration file of the previous Section 6.2.2. A single cell-veto event handler realizes the set of Coulomb four-body factors that relate to non-excluded cells with respect to the induced active dipole's cell. While in the previous section the number of event handlers using a cell-based bounding potential must exceed the number of target dipoles in non-excluded cells, here a single cell-veto event handler is sufficient for an arbitrary number of target dipoles in different non-excluded cells. This allows to implement ECMC with a complexity of $\mathcal{O}(1)$ per event.

The configuration file resembles that of Section 6.2.2 except that the `coulomb_cell_bounding` event handler is replaced by a `coulomb_cell_veto` event handler (using the cell-veto event handler of Section 4.1.5). Also a different tagger is used because the in-state of the cell-veto event handler only consists of the branch of the induced active dipole (see the discussion in Section 6.1.3 where the cell-veto algorithm was used in order to simulate two charged point masses).

6.2.4 Coulomb Dipole Factors, Alternating Movement Modes

In contrast to all other simulations of this chapter with only a single independent active point mass at any time, the `dipole_motion.ini` configuration file of this section alternates between two movement modes: In the leaf mode, one of the four point masses is independent active (the name results from point masses being stored on the leaf nodes of the trees in the tree state handler). In the root mode, the point masses of one dipole move as a rigid block. Hence, the entire dipole is independent active (see Fig. 6.3). Only considering the root mode does not yield an ergodic Markov chain, as discussed in Section 4.2.5. The orientation and shape of any dipole would remain unchanged throughout the run.

Since the dipoles are represented as trees in JF-V1.0, both modes are easily implemented. In the leaf mode, the Coulomb four-body factor of all four point masses is realized by a `coulomb_leaf` event handler that uses the same bounding potential as the `coulomb_nearby` event handlers of the previous Sections 6.2.2 and 6.2.3. The harmonic and repulsive pair factor are also treated in the same manner as before. The root mode, in turn, is patterned after the simulation of two point masses (see Section 6.1.1). Both the four-body Coulomb factor and the repulsive pair factor yield an effective two-dipole factor realized by the event handlers of Section 4.1.6. Confirmed events of these just switch the independent active dipole. (The inner-dipole harmonic factor potential is constant and therefore not considered in root mode.) The create and trash decision is straightforward because all factor events induce the trashing of factor events in the scheduler and their following recomputation.

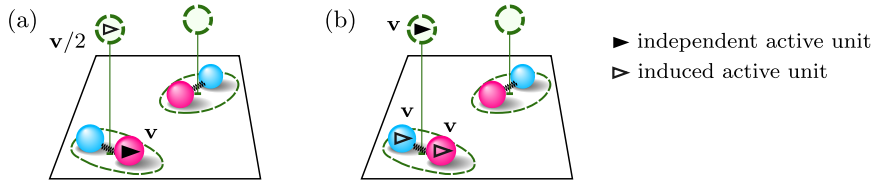


Figure 6.3: Two alternating movement modes used to simulate two finite-size dipoles. (a): In the leaf mode, a single leaf unit (which corresponds to a single point mass) is independent active with the velocity $\mathbf{v}$. This induces a movement of the corresponding dipole barycenter with $\mathbf{v}/2$. (b): In the root mode, one of the two root units (which corresponds to a dipole barycenter) is independent active with velocity $\mathbf{v}$. Both of its leaf units then have an induced velocity $\mathbf{v}$. This figure has been published in [39] (see publication in Appendix B).

The switches between the leaf and root modes are formulated as `leaf_to_root` and `root_to_leaf` events, respectively. For this, two instances of the mode-switching event handler of Section 4.2.5 with different mode-switching directions are used. The tag lists of the corresponding taggers trash factor events of the old mode, and entail the computation of factor events for the new mode. Also, the event responsible for the next mode switch is computed afterwards.

After a mode-switching event, all taggers connected to factor types of the new mode are additionally activated, whereas the ones of the old mode are deactivated (as introduced in Section 3.5, deactivated taggers do not generate any in-state identifiers). By this, the end-of-chain event handler can still create events in equally spaced intervals: After an end-of-chain event, all factor events are recomputed which is encoded in the corresponding tag list. However, it is not clear which mode is currently active and, e.g., whether a `coulomb_leaf` or a `coulomb_root` event has to be computed. The create tag list just contains both tags instead and exploits that only one of the corresponding taggers is active.

Introducing different movement modes may influence the mixing dynamics of ECMC and lead to a faster decorrelation. (Note however that this was not yet studied in detail). At the same time, the non-ergodic moves might necessitate a special care on when to sample an observable. The configuration file of this section samples after fixed sampling durations (as all other configuration files of this chapter). Hence, sample values are also computed during the non-ergodic parts of the Markov chain. (This could be a reason for the observed increased standard errors of this method in Fig. 6.2.)

6.3 Interacting Water Molecules (SPC/Fw Model)

The configuration files of this section implement the ECMC sampling of the Boltzmann distribution for two water molecules using the SPC/FW model [82] that was previously studied with ECMC [7].[3] A single water molecule is represented by a composite point object with three charged point masses in the tree state handler. One of the point masses is positively charged and represents the oxygen. The two others are negatively charged and represent the hydrogens.

The SPC/Fw model uses the displaced-even-power potential of Section 4.1.1.3 with the parameters $p_{\text{DEPP}} = 2$, $r_0 = 1.012\,\text{Å}$, and $k_{\text{DEPP}} = 529.581\,\text{kcal}\,\text{mol}^{-1}\,\text{Å}^{-2}$ between the hydrogens and oxygens within a water molecule. The three point masses of a water molecule interact via the bending potential of Section 4.1.1.5 with the parameters $k_{\text{Bending}} = 75.90\,\text{kcal}\,\text{mol}^{-1}\,\text{rad}^{-2}$ and $\phi_0 = 113.24°$, which

[3] The used configuration files are located in the `src/config_files/2018_JCP_149_064113/water` directory of JF-V1.0.

induces fluctuations around the given equilibrium angle ϕ_0 in the samples of the simulation. Moreover, two oxygens in different water molecules interact via the Lennard-Jones potential of Section 4.1.1.2 with the parameters $k_{\mathrm{LJ}} = 0.6217012\,\mathrm{kcal\,mol^{-1}}$ and $\sigma_{\mathrm{LJ}} = 3.165492\,\text{Å}$. The Lennard-Jones potential is truncated beyond 9.0 Å. Finally, point masses in different water molecules interact via the merged-image Coulomb potential of Section 4.1.1.4. Here, the charges of the hydrogens are chosen as $0.41e$, and the ones of the oxygens as $-0.82e$ (with e as the elementary charge). All simulations of this section were carried out in a three-dimensional cubic simulation box with side length $L = 10\,\text{Å}$ and periodic boundary conditions. The temperature was chosen as 300 K (i.e., $\beta = 1.679\,\mathrm{mol\,kcal^{-1}}$).

In all of the configuration files one of the six point masses is active with a velocity parallel to one of the Cartesian coordinate axes. (Of course, one could set up a similar rigid movement of entire water molecules as in Section 6.2.4). The `TwoLeafUnitEventHandler` of Section 4.1.2 realizes a single harmonic pair factor with the displaced-even-power-law factor potential. The corresponding tagger then relies on two of these event handlers because if one of the oxygens is independent active, two harmonic pair factors have to be considered. The bending three-body factor with the bending factor potential is realized in an event handler that dynamically constructs a piecewise constant bounding potential (see Section 4.1.5). The event handler is initialized with a lifting scheme to carry out the lifting move in its out-state computation. (Note that the lifting probabilities are uniquely determined in this case because there are only three point masses involved.)

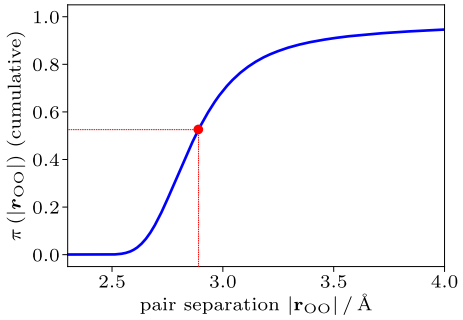
The following sections discuss how the remaining Coulomb and Lennard-Jones factors are treated. The generation of the in-state identifiers in the taggers and the determination of which events should be trashed and recomputed is very similar to the previous simulations of two finite-size dipoles in Section 6.2. This is therefore not explained in detail again. Statistically equivalent output is obtained for pair factors for the Coulomb and inverted Lennard-Jones potentials in Section 6.3.1. In Section 6.3.2, molecular six-body Coulomb factors are considered and the Lennard-Jones factor is associated with a cell-based bounding potential. Section 6.3.3 implements the cell-veto algorithm for the molecular Coulomb factor and inverts the Lennard-Jones potential. The final Section 6.3.4 implements the cell-veto algorithm for both the six-body molecular Coulomb factor and the Lennard-Jones pair factor between the oxygens. All of the simulations sample the minimum separation between the two oxygens (see Fig. 6.4 for a comparison of the results of the different simulations).

It is noteworthy that all the following simulations only considered the Lennard-Jones interaction between the nearest images of the oxygens in the two water molecules. (That is because the event handlers explicitly compute the minimum separation.) The simulations do therefore not use the aforementioned truncation after 9 Å. It was tested that this does not lead to considerable changes of the results. (It would surely be possible to implement such a truncation in JF. This would require new event handlers. For the sake of simplicity, these are not included in this thesis and JF-V1.0.)

6.3.1 Coulomb Pair Factors, Lennard-Jones Inverted

In the `coulomb_power_bounded_lj_inverted.ini` configuration file the Lennard-Jones pair factor is realized by the `TwoLeafUnitEventHandler` of Section 4.1.2, which internally uses the inverted Lennard-Jones potential. The event handler only computes an event if one of the involved oxygens is independent active.

The three Coulomb pair factors that involve the currently independent active point mass are realized in three instances of the `TwoLeafUnitBoundingPotentialEventHandler` of Section 4.1.3. This event handler uses both the merged-image Coulomb potential and its bounding potential of Section 4.1.1.4.



method	$\pi(\lvert\mathbf{r}_{OO}\rvert < 2.9\,\text{Å})$
①	0.5382 ± 0.0003
②	0.5385 ± 0.0003
③	0.5390 ± 0.0002
④	0.5400 ± 0.0010
⑤	0.5398 ± 0.0010

Figure 6.4: Cumulative histogram $\pi(|\mathbf{r}_{OO}|)$ of the minimum pair separations between the nearest images of the two oxygens in different water molecules using the SPC/Fw model [82]. The simulations used a three-dimensional cubic simulation box with periodic boundary conditions. Moreover, $\pi(|\mathbf{r}_{OO}| < 2.9\,\text{Å})$ including standard errors is shown for each simulation method. The obtained values and their standard errors agree well with the weighted average of all values within at most two times their respective standard errors. ①: Reversible Markov-chain Monte Carlo. ②: Method of Section 6.3.1. ③: Method of Section 6.3.2. ④: Method of Section 6.3.3. ⑤: Method of Section 6.3.4. A modified version of this figure has been published in [39] (see publication in Appendix B). Note especially that the published version of this figure contained a typo in the result of the first method. This is going to be corrected in the next version available online.

6.3.2 Molecular Coulomb Factors, Lennard-Jones Cell-Bounded

The `coulomb_power_bounded_lj_cell_bounded.ini` configuration file implements molecular six-body Coulomb factors by summing the involved Coulomb pair potentials. This factor is therefore very similar to the Coulomb dipole factor of the Sections 6.2.2 and 6.2.3 and is realized by an event handler that is analogous to the `coulomb_nearby` event handler in these sections.

Although the Lennard-Jones potential is inverted, it is associated with a cell-based bounding potential in this section. For this, a cell-occupancy system that only tracks the oxygen identifiers is set up. This is achieved by introducing an indicator charge that is only non-zero for the oxygens. The cell-occupancy system is then initialized with the name of this charge, which leads to the fact that only the oxygens are tracked. The derivative bounds used in the cell-based bounding potential are computed by an inner-point estimator (see Section 4.1.1.6). Naturally, the usage of a cell-based bounding potential induces nearby, surplus and cell-boundary events.

6.3.3 Molecular Coulomb Cell-Veto, Lennard-Jones Inverted

The `coulomb_cell_veto_lj_inverted.ini` configuration file uses the same factors as the previous Section 6.3.2. The Lennard-Jones pair factor is treated like in Section 6.3.1. As a preliminary step toward the treatment of several interactions via the cell-veto algorithm in the following section, the molecular Coulomb factors are realized in a cell-veto event handler. Hence, a cell-occupancy system in the tag activator tracks the barycenters of the water molecules. The cell-based derivative bounds are computed in the dipole-inner-point estimator of Section 4.1.1.6. The surplus and nearby molecular Coulomb factors are treated in a similar way as in the corresponding simulation for finite-size dipoles in Section 6.2.3. (Similar to Section 6.1.3, the simulation is slowed down due to a lot of rejected cell-veto events which increases the standard error in Fig. 6.4. That is because the simulation was not run for longer times to keep the simulation times in a reasonable limit.)

6.3.4 Molecular Coulomb Cell-Veto, Lennard-Jones Cell-Veto

The final `coulomb_cell_veto_lj_cell_veto.ini` configuration file uses, for illustration purposes, two cell-occupancy systems and two cell-veto event handlers for both the molecular Coulomb factor and the Lennard-Jones pair factor. The cell-occupancy systems and estimators are chosen as in the previous two sections. Setting up the cell-veto algorithm twice with two different cell-occupancy systems also implies two sets of nearby and surplus event handlers, as well as two cell-boundary event handlers. This example showcases the flexibility of the activator, which can use several different cell-occupancy systems during one simulation. If desired, the internal states can even use different underlying cell systems. (The standard error in Fig. 6.4 is again increased due to the high number of rejected cell-veto events, similar to the previous Section 6.3.3).

Note, as a final remark, that the strengths of the cell-veto algorithm are exploited when the number of water molecules is increased. Furthermore, the simulations using the cell-veto algorithm in this chapter are considerably sped-up if no time-slicing is performed after unconfirmed events (see the discussion in Section 3.7). This acceleration of the corresponding simulations were already observed in first enhancements of JF-V1.0 that resign from the time-slicing after unconfirmed events (and therefore do not lead to unnecessary trashing and recomputations of events). These enhancements are going to be included in the next versions of JF.

CHAPTER 7

Conclusion

This thesis started with an introduction into the event-chain Monte Carlo (ECMC) algorithm and demonstrated how it constructs an irreversible rejection-free continuous-time Markov chain. For this, the factorized Metropolis filter, the lifting framework, and the event-driven implementation were discussed. Additionally, the cell-veto algorithm was introduced. This algorithm allows to compute the next event that interrupts the piecewise non-interacting time-evolution of ECMC in $\mathcal{O}(1)$ operations.

Due to its irreversible nature, ECMC simulations were shown to often equilibrate faster than simulations that construct a reversible Markov chain. In particular, ECMC simulations solved the puzzle about the precise nature of the melting phase transition of two-dimensional hard disks after 49 years. Previous studies using both molecular dynamics simulations or other Monte Carlo approaches were not able to achieve that. This shows the strong potential of the ECMC algorithm and justifies the expectation that it may prove useful for simulations of all-atom systems including the long-range Coulomb potential: a domain that has traditionally been reserved to molecular dynamics. ECMC may also help to gain new insights in other areas, e.g., spin models or problems in quantum-field theory.

The strong potential of the ECMC algorithm motivates the implementation of a general-purpose application that implements it in a way that it can treat a wide range of different systems. This is precisely the intention of the JeLLyFysh (JF) application. Its design was introduced in this thesis and in the complementing publication [39] (see publication in Appendix B). Built on the mediator design pattern, the application systematically formulates the entire time evolution of ECMC in terms of events. Here, both factors and pseudo-factors yield events. Factor events are required by ECMC to satisfy the global-balance condition. In contrast, events of the newly introduced pseudo-factors have no incidence on this condition. Instead, they permit JF to represent an entire run of the application in terms of events and include, for example, start-of-run, end-of-run, sampling, and end-of-chain events (which are necessary to construct an ergodic Markov chain).

The mediator of JF contains the main iteration loop of ECMC that proceeds from one event to the next. For this, it uses the different elements of JF: Event handlers compute events, whose candidate event times are compared in the scheduler. Only the out-state of the event with the shortest candidate event time changes the global state, which is stored in the state handler. The activator determines the event handlers that have to compute events at the beginning of each iteration. Also, it yields the events which became invalid at the end of an iteration after the global state was changed. The input-output handler inserts information into the application and creates output. In JF, all of the mentioned elements are defined in an abstract way to ensure a general implementation of the time evolution of ECMC.

Since JF is an open-source project that is publicly available on GitHub, users can study, execute, modify, and distribute the fully documented source code. Modifications can be fed back into the project and contributions are encouraged. By this, JF is intended to grow into a widely used basis code that hopefully becomes useful for researchers in different fields of computational science.

In order to showcase the flexibility and potential of the application, this thesis explained in detail how the different elements are implemented for all-atom simulations in the first version of the application JF-V1.0. (Also, the thesis gave hints which parts can even be reused for other models.) This allowed for simulations of long-range interacting systems in a number of worked-out examples ranging from charged atoms to dipoles to water molecules.

For the first version of the application, JeLLyFysh-Version1.0 (JF-V1.0), consistency has been the main concern and the implemented code has not yet been optimized. It is entirely written in the Python language and is compatible with any interpreter that supports Python 3.5 and higher. (This includes, of course, the appropriate versions of CPython, but also PyPy in version 7 and higher [83]. PyPy uses a just-in-time compiler that leads to significant speed improvements over CPython in the presented simulation examples.) Considerable speed-up can certainly be obtained by rewriting time-consuming parts of the application in compiled C/C++ codes, which includes in particular the methods of the potentials that are included in JF-V1.0.

Further tasks for future versions of the JF application were mentioned at the appropriate places throughout this thesis. A major priority is certainly that unconfirmed events are unnecessarily time-sliced, which leads to superfluous trashing and recomputing of candidate events. Moreover, future versions should include the presented straightforward implementation of an arbitrary number $n_{\mathrm{ac}} > 1$ of simultaneously active particles only taking part in distance-dependent interaction.

An arbitrary number $n_{\mathrm{ac}} > 1$ will enable full parallel implementations of ECMC on multiprocessor machines. The details of the parallel algorithm, however, have still to be studied further. As a first step to multi-process ECMC, JF-V1.0 already implements the parallel computation of events in separate processes, which is, however, at present rather slow due to the required inter-process communication. The outstanding challenge of JF in its future versions is therefore to bring the full power of parallelization and of multi-process ECMC to real-world applications.

Bibliography

[1] N. Metropolis et al., *Equation of State Calculations by Fast Computing Machines*, The Journal of Chemical Physics **21** (1953) 1087, URL: https://doi.org/10.1063/1.1699114 (cit. on pp. 1, 7–9, 14).

[2] B. J. Alder and T. E. Wainwright, *Phase Transition for a Hard Sphere System*, The Journal of Chemical Physics **27** (1957) 1208, URL: https://doi.org/10.1063/1.1743957 (cit. on pp. 1, 2, 14).

[3] C. Kittel, *Introduction to Solid State Physics*, 8th ed., John Wiley & Sons, 2004, ISBN: 9780471415268 (cit. on p. 1).

[4] J. Bardeen, L. N. Cooper, and J. R. Schrieffer, *Theory of Superconductivity*, Phys. Rev. **108** (1957) 1175, URL: https://link.aps.org/doi/10.1103/PhysRev.108.1175 (cit. on p. 1).

[5] M. Toda, R. Kubo, and N. Saito, *Statistical Physics I, Equilibrium Statistical Mechanics*, 1st ed., Springer, 1983, ISBN: 9783642967009 (cit. on pp. 1, 13).

[6] W. Krauth, *Statistical Mechanics: Algorithms and Computations*, 1st ed., Oxford University Press, 2006, ISBN: 9780198515357 (cit. on pp. 1, 6, 13–15).

[7] M. F. Faulkner et al., *All-atom computations with irreversible Markov chains*, The Journal of Chemical Physics **149** (2018) 064113, URL: https://doi.org/10.1063/1.5036638 (cit. on pp. 1–3, 5, 15, 17, 19, 23–25, 28, 52–54, 59, 67, 69, 71, 72, 74).

[8] E. P. Bernard, W. Krauth, and D. B. Wilson, *Event-chain Monte Carlo algorithms for hard-sphere systems*, Phys. Rev. E **80** (2009) 056704, URL: https://link.aps.org/doi/10.1103/PhysRevE.80.056704 (cit. on pp. 1, 27).

[9] M. Michel, S. C. Kapfer, and W. Krauth, *Generalized event-chain Monte Carlo: Constructing rejection-free global-balance algorithms from infinitesimal steps*, The Journal of Chemical Physics **140** (2014) 054116, URL: https://doi.org/10.1063/1.4863991 (cit. on pp. 1, 2, 15, 23).

[10] Y. Nishikawa et al., *Event-chain algorithm for the Heisenberg model: Evidence for $z \simeq 1$ dynamic scaling*, Phys. Rev. E **92** (2015) 063306, URL: https://link.aps.org/doi/10.1103/PhysRevE.92.063306 (cit. on pp. 1, 2).

[11] S. C. Kapfer and W. Krauth, *Irreversible Local Markov Chains with Rapid Convergence towards Equilibrium*, Phys. Rev. Lett. **119** (2017) 240603, URL: https://link.aps.org/doi/10.1103/PhysRevLett.119.240603 (cit. on p. 1).

[12] Z. Lei and W. Krauth, *Irreversible Markov chains in spin models: Topological excitations*, EPL (Europhysics Letters) **121** (2018) 10008, URL: https://doi.org/10.1209%2F0295-5075%2F121%2F10008 (cit. on pp. 1, 2).

[13] Z. Lei and W. Krauth, *Mixing and perfect sampling in one-dimensional particle systems*, EPL (Europhysics Letters) **124** (2018) 20003, URL: https://doi.org/10.1209%2F0295-5075%2F124%2F20003 (cit. on pp. 1, 24, 27).

[14] Z. Lei, W. Krauth, and A. C. Maggs, *Event-chain Monte Carlo with factor fields*, Phys. Rev. E **99** (2019) 043301, URL: https://link.aps.org/doi/10.1103/PhysRevE.99.043301 (cit. on pp. 1, 2, 17).

[15] E. P. Bernard and W. Krauth, *Two-Step Melting in Two Dimensions: First-Order Liquid-Hexatic Transition*, Phys. Rev. Lett. **107** (2011) 155704, URL: https://link.aps.org/doi/10.1103/PhysRevLett.107.155704 (cit. on pp. 1, 14, 15, 30).

[16] M. Engel et al., *Hard-disk equation of state: First-order liquid-hexatic transition in two dimensions with three simulation methods*, Phys. Rev. E **87** (2013) 042134, URL: https://link.aps.org/doi/10.1103/PhysRevE.87.042134 (cit. on p. 1).

[17] J. Harland et al., *Event-chain Monte Carlo algorithms for three- and many-particle interactions*, EPL (Europhysics Letters) **117** (2017) 30001, URL: https://doi.org/10.1209%2F0295-5075%2F117%2F30001 (cit. on pp. 2, 15, 22).

[18] E. A. J. F. Peters and G. de With, *Rejection-free Monte Carlo sampling for general potentials*, Phys. Rev. E **85** (2012) 026703, URL: https://link.aps.org/doi/10.1103/PhysRevE.85.026703 (cit. on pp. 2, 24, 25).

[19] M. Michel, *Irreversible Markov chains by the factorized Metropolis filter: algorithms and applications in particle systems and spin models*, PhD thesis: PSL Research University, 2016, URL: http://www.theses.fr/2016PSLEE039 (cit. on pp. 2, 11).

[20] S. C. Kapfer and W. Krauth, *Two-Dimensional Melting: From Liquid-Hexatic Coexistence to Continuous Transitions*, Phys. Rev. Lett. **114** (2015) 035702, URL: https://link.aps.org/doi/10.1103/PhysRevLett.114.035702 (cit. on p. 2).

[21] M. Michel, J. Mayer, and W. Krauth, *Event-chain Monte Carlo for classical continuous spin models*, EPL (Europhysics Letters) **112** (2015) 20003, URL: https://doi.org/10.1209%2F0295-5075%2F112%2F20003 (cit. on p. 2).

[22] M. Hasenbusch and S. Schaefer, *Testing the event-chain algorithm in asymptotically free models*, Phys. Rev. D **98** (2018) 054502, URL: https://link.aps.org/doi/10.1103/PhysRevD.98.054502 (cit. on p. 2).

[23] F. Chen, L. Lovász, and I. Pak, *Lifting Markov Chains to Speed Up Mixing*, *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, ACM, 1999 275, URL: http://doi.acm.org/10.1145/301250.301315 (cit. on pp. 2, 9, 19).

[24] P. Diaconis, S. Holmes, and R. M. Neal, *Analysis of a nonreversible Markov chain sampler*, Ann. Appl. Probab. **10** (2000) 726, URL: https://doi.org/10.1214/aoap/1019487508 (cit. on pp. 2, 9, 19).

[25] B. J. Alder and T. E. Wainwright, *Studies in Molecular Dynamics. I. General Method*, The Journal of Chemical Physics **31** (1959) 459, URL: https://doi.org/10.1063/1.1730376 (cit. on p. 2).

[26] M. N. Bannerman and L. Lue, *Exact on-event expressions for discrete potential systems*, The Journal of Chemical Physics **133** (2010) 124506, URL: https://doi.org/10.1063/1.3486567 (cit. on p. 2).

[27] M. H. A. Davis, *Piecewise-Deterministic Markov Processes: A General Class of Non-Diffusion Stochastic Models*, Journal of the Royal Statistical Society. Series B (Methodological) **46** (1984) 353, URL: http://www.jstor.org/stable/2345677 (cit. on p. 2).

[28] J. Bierkens et al., *Piecewise deterministic Markov processes for scalable Monte Carlo on restricted domains*, Statistics & Probability Letters **136** (2018) 148, URL: http://www.sciencedirect.com/science/article/pii/S016771521830066X (cit. on p. 2).

[29] S. C. Kapfer and W. Krauth, *Cell-veto Monte Carlo algorithm for long-range systems*, Phys. Rev. E **94** (2016) 031302, URL: https://link.aps.org/doi/10.1103/PhysRevE.94.031302 (cit. on pp. 2, 17, 27–29).

[30] M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, 1st ed., Clarendon Press, 1989, ISBN: 9780198556459 (cit. on p. 2).

[31] F. Ding et al., *Ab Initio Folding of Proteins with All-Atom Discrete Molecular Dynamics*, Structure **16** (2008) 1010, URL: http://www.sciencedirect.com/science/article/pii/S0969212608001780 (cit. on p. 2).

[32] E. A. Proctor, F. Ding, and N. V. Dokholyan, *Discrete molecular dynamics*, Wiley Interdisciplinary Reviews: Computational Molecular Science **1** (2011) 80, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.4 (cit. on p. 2).

[33] S. C. Kapfer and W. Krauth, *Sampling from a polytope and hard-disk Monte Carlo*, Journal of Physics: Conference Series **454** (2013) 012031, URL: https://doi.org/10.1088%2F1742-6596%2F454%2F1%2F012031 (cit. on p. 3).

[34] T. A. Kampmann, H.-H. Boltz, and J. Kierfeld, *Parallelized event chain algorithm for dense hard sphere and polymer systems*, Journal of Computational Physics **281** (2015) 864, URL: http://www.sciencedirect.com/science/article/pii/S0021999114007475 (cit. on p. 3).

[35] S. Miller and S. Luding, *Event-driven molecular dynamics in parallel*, Journal of Computational Physics **193** (2004) 306, URL: http://www.sciencedirect.com/science/article/pii/S0021999103004339 (cit. on p. 3).

[36] M. C. Herbordt, M. A. Khan, and T. Dean, *Parallel Discrete Event Simulation of Molecular Dynamics Through Event-Based Decomposition*, *2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009 129, URL: https://ieeexplore.ieee.org/document/5200020 (cit. on p. 3).

[37] M. A. Khan and M. C. Herbordt, *Parallel discrete molecular dynamics simulation with speculation and in-order commitment*, Journal of Computational Physics **230** (2011) 6563, URL: http://www.sciencedirect.com/science/article/pii/S0021999111002968 (cit. on p. 3).

[38] A. G. Lowndes, *Percentage of Water in Jelly-Fish*, Nature **150** (1942) 234, URL: https://doi.org/10.1038/150234b0 (cit. on p. 3).

[39] P. Höllmer et al., *JeLLyFysh-Version1.0 – a Python application for all-atom event-chain Monte Carlo*, arXiv e-prints (2019), arXiv: 1907.12502 [physics.comp-ph] (cit. on pp. 3, 24, 32–34, 37, 38, 42–44, 48, 62, 65, 66, 69, 71, 74, 76, 79).

[40] E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Prentice Hall, 1994, ISBN: 9780201633610 (cit. on pp. 3, 31).

[41] N. Metropolis and S. Ulam, *The Monte Carlo Method*, Journal of the American Statistical Association **44** (1949) 335, PMID: 18139350, URL: https://www.tandfonline.com/doi/abs/10.1080/01621459.1949.10483310 (cit. on p. 6).

[42] M. Loève, *Probability Theory I*, 4th ed., Springer, 1977, ISBN: 9780387902104 (cit. on p. 6).

[43] J. von Neumann, *Various Techniques Used in Connection with Random Digits*, *Monte Carlo Method*, ed. by A. S. Householder, G. E. Forsythe, and H. H. Germond, National Bureau of Standards Applied Mathematics Series **12**, US Government Printing Office, 1951 (cit. on p. 7).

[44] L. Devroye, *Non-Uniform Random Variate Generation*, 1st ed., Springer, 1986, ISBN: 9780387963051 (cit. on pp. 7, 25).

[45] A. A. Markov, *Extension of the limit theorems of probability theory to a sum of variables connected in a chain*, (1907), reprinted in Appendix B of: R. Howard, *Dynamic Probabilistic Systems, volume 1: Markov Chains*, John Wiley and Sons, 1971 (cit. on p. 8).

[46] D. P. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, 1st ed., Cambridge University Press, 2000, ISBN: 9780521653664 (cit. on pp. 8–10, 13).

[47] N. G. Van Kampen, *Stochastic Processes in Physics and Chemistry*, 3rd ed., North Holland, 2007, ISBN: 9780444529657 (cit. on p. 8).

[48] D. A. Levin, Y. Peres, and E. L. Wilmer, *Markov Chains and Mixing Times*, 1st ed., American Mathematical Society, 2008, ISBN: 9780821847398 (cit. on pp. 9, 11).

[49] W. K. Hastings, *Monte Carlo sampling methods using Markov chains and their applications*, Biometrika **57** (1970) 97, URL: https://doi.org/10.1093/biomet/57.1.97 (cit. on p. 9).

[50] R. H. Swendsen and J.-S. Wang, *Nonuniversal critical dynamics in Monte Carlo simulations*, Phys. Rev. Lett. **58** (1987) 86, URL: https://link.aps.org/doi/10.1103/PhysRevLett.58.86 (cit. on p. 9).

[51] U. Wolff, *Collective Monte Carlo Updating for Spin Systems*, Phys. Rev. Lett. **62** (1989) 361, URL: https://link.aps.org/doi/10.1103/PhysRevLett.62.361 (cit. on p. 9).

[52] C. Dress and W. Krauth, *Cluster algorithm for hard spheres and related systems*, Journal of Physics A: Mathematical and General **28** (1995) L597, URL: https://doi.org/10.1088%2F0305-4470%2F28%2F23%2F001 (cit. on p. 9).

[53] P. Gustafson, *A guided walk Metropolis algorithm*, Statistics and Computing **8** (1998) 357, URL: https://doi.org/10.1023/A:1008880707168 (cit. on p. 9).

[54] S. Duane et al., *Hybrid Monte Carlo*, Physics Letters B **195** (1987) 216, URL: http://www.sciencedirect.com/science/article/pii/037026938791197X (cit. on p. 9).

[55] R. M. Neal, *Technical Report No. 9508*, Department of Statistics, University of Toronto (1995) (cit. on p. 9).

[56] E. P. Bernard, C. Chanal, and W. Krauth, *Damage spreading and coupling in Markov chains*, EPL (Europhysics Letters) **92** (2010) 60004, URL: https://doi.org/10.1209%2F0295-5075%2F92%2F60004 (cit. on p. 11).

[57] H. Flyvbjerg and H. G. Petersen, *Error estimates on averages of correlated data*, The Journal of Chemical Physics **91** (1989) 461, URL: https://doi.org/10.1063/1.457480 (cit. on pp. 11–13).

[58] A. Sokal, *Monte Carlo Methods in Statistical Mechanics: Foundations and New Algorithms*, *Functional Integration*, ed. by C. DeWitt-Morette, P. Cartier, and A. Folacci, NATO ASI Series (Series B: Physics) **361**, Springer, 1997, ISBN: 9781489903211 (cit. on pp. 12, 13).

[59] K. Binder, *Monte Carlo Investigations of Phase Transitions and Critical Phenomena*, *Phase Transitions and Critical Phenomena*, ed. by C. Domb and M. S. Green, vol. 5B, Academic, 1976, ISBN: 9780122203510 (cit. on p. 13).

[60] J. G. Propp and D. B. Wilson, *Exact Sampling with Coupled Markov Chains and Applications to Statistical Mechanics*, Random Structures & Algorithms **9** (1996) 223, URL: https://doi.org/10.1002/(SICI)1098-2418(199608/09)9:1/2%3C223::AID-RSA14%3E3.0.CO;2-O (cit. on p. 13).

[61] M. N. Rosenbluth and A. W. Rosenbluth, *Further Results on Monte Carlo Equations of State*, The Journal of Chemical Physics **22** (1954) 881, URL: https://doi.org/10.1063/1.1740207 (cit. on p. 14).

[62] W. W. Wood and J. D. Jacobson, *Preliminary Results from a Recalculation of the Monte Carlo Equation of State of Hard Spheres*, The Journal of Chemical Physics **27** (1957) 1207, URL: https://doi.org/10.1063/1.1743956 (cit. on pp. 14, 15).

[63] W. G. Hoover and F. H. Ree, *Melting Transition and Communal Entropy for Hard Spheres*, The Journal of Chemical Physics **49** (1968) 3609, URL: https://doi.org/10.1063/1.1670641 (cit. on p. 14).

[64] B. J. Alder and T. E. Wainwright, *Phase Transition in Elastic Disks*, Phys. Rev. **127** (1962) 359, URL: https://link.aps.org/doi/10.1103/PhysRev.127.359 (cit. on p. 14).

[65] N. D. Mermin and H. Wagner, *Absence of Ferromagnetism or Antiferromagnetism in One- or Two-Dimensional Isotropic Heisenberg Models*, Phys. Rev. Lett. **17** (1966) 1133, URL: https://link.aps.org/doi/10.1103/PhysRevLett.17.1133 (cit. on p. 14).

[66] N. D. Mermin, *Crystalline Order in Two Dimensions*, Phys. Rev. **176** (1968) 250, URL: https://link.aps.org/doi/10.1103/PhysRev.176.250 (cit. on p. 14).

[67] J. E. Jones and S. Chapman, *On the determination of molecular fields. — II. From the equation of state of a gas*, Proceedings of the Royal Society of London A **106** (1924) 463, URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1924.0082 (cit. on p. 16).

[68] A. Bortz, M. Kalos, and J. Lebowitz, *A new algorithm for Monte Carlo simulation of Ising spin systems*, Journal of Computational Physics **17** (1975) 10, URL: http://www.sciencedirect.com/science/article/pii/0021999175900601 (cit. on p. 24).

[69] P. A. W. Lewis and G. S. Shedler, *Simulation of nonhomogeneous poisson processes by thinning*, Naval Research Logistics Quarterly **26** (1979) 403, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800260304 (cit. on p. 26).

[70] S. N. Chiu et al., *Stochastic Geometry and its Applications*, 3rd ed., John Wiley & Sons, 2013, ISBN: 9780470664810 (cit. on p. 26).

[71] R. F. B. Weigel, *Equilibration of Orientational Order in Hard Disks via Arcuate Event-Chain Monte Carlo*, MA thesis: Friedrich-Alexander-Universität Erlangen-Nürnberg, 2018, URL: https://theorie1.physik.uni-erlangen.de/media/pdf/theses/2018_ma_roweigel.pdf (cit. on p. 27).

[72] A. J. Walker, *An Efficient Method for Generating Discrete Random Variables with General Distributions*, ACM Trans. Math. Softw. **3** (1977) 253, URL: http://doi.acm.org/10.1145/355744.355749 (cit. on p. 28).

[73] E. Bernard, *Algorithms and applications of the Monte Carlo method: Two-dimensional melting and perfect sampling*, PhD thesis: Université Pierre et Marie Curie - Paris VI, 2011, URL: https://tel.archives-ouvertes.fr/tel-00637330 (cit. on p. 29).

[74] T. H. Cormen et al., *Introduction to Algorithms*, 2nd ed., MIT Press, 2001, ISBN: 9780262032933 (cit. on p. 39).

[75] *Crystallography: Protein Data Bank*, Nature New Biology **233** (1971) 223, URL: https://doi.org/10.1038/newbio233223b0 (cit. on p. 45).

[76] N. Michaud-Agrawal et al., *MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations*, Journal of Computational Chemistry **32** (2011) 2319, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/jcc.21787 (cit. on p. 45).

[77] R. Gowers et al., *MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations*, *Proceedings of the 15th Python in Science Conference*, ed. by S. Benthall and S. Rostrup, SciPy, 2016, URL: https://doi.org/10.25080/majora-629e541a-00e (cit. on p. 45).

[78] M. M. McKerns and M. A. G. Aivazis, *Pathos: a framework for heterogeneous computing*, 2010, URL: http://trac.mystic.cacr.caltech.edu/project/pathos (cit. on pp. 46, 66).

[79] M. M. McKerns et al., *Building a Framework for Predictive Science*, *Proceedings of the 10th Python in Science Conference*, ed. by S. van der Walt and J. Millman, SciPy, 2011 (cit. on pp. 46, 66).

[80] S. W. de Leeuw et al., *Simulation of electrostatic systems in periodic boundary conditions. I. Lattice sums and dielectric constants*, Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences **373** (1980) 27, URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1980.0135 (cit. on p. 52).

[81] D. Frenkel and B. Smit, *Understanding Molecular Simulation, From Algorithms to Applications*, 2nd ed., Academic Press, 2002, ISBN: 9780122673511 (cit. on p. 52).

[82] Y. Wu, H. L. Tepper, and G. A. Voth, *Flexible simple point-charge water model with improved liquid-state properties*, The Journal of Chemical Physics **124** (2006) 024503, URL: https://doi.org/10.1063/1.2136877 (cit. on pp. 74, 76).

[83] C. F. Bolz et al., *Tracing the Meta-level: PyPy's Tracing JIT Compiler*, *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ACM, 2009, ISBN: 978-1-60558-541-3, URL: http://doi.acm.org/10.1145/1565824.1565827 (cit. on p. 80).

APPENDIX A

List of Implemented Taggers

This appendix lists all taggers that are implemented in JF-V1.0. These are used in the tag activator (see Section 3.5). Furthermore, an explanation on how these taggers generate the in-state identifiers for their event handlers in the `yield_identifiers_send_event_time` method is given. The only argument of this method is the extracted active global state, that is, the part of the global state that appears in the global lifting state (see Section 3.2).

The taggers of JF-V1.0 are specifically implemented for the tree state handler. The active global state is given by root cnodes of branches, where all cnodes in the branches contain an active unit. All unit identifiers that appear in the active global state are unique. Taggers may use an internal state. In JF-V1.0, these are given by cell-occupancy systems (see Section 3.5.1).

- `NoInStateTagger`:
  Contains a single event handler and returns `None` as the in-state identifier for it. This implies that the event handlers' `send_event_time` method requires no arguments. An example for such an event handler is the end-of-chain event handler of Section 4.2.3.

- `FactorTypeMapInStateTagger`:
  This special tagger can only be used if the tree state handler contains a state that consists of an arbitrary number of identical trees representing composite point objects of height at most two (see Section 5.1). The tagger reads identifier sets of factors between point masses in two different composite point objects from a file, and extrapolates these to factors between all composite point objects. (The identifier sets correspond to the index sets I_M of factors introduced in Section 2.3.) When the in-state identifiers are requested, all identifier sets that include a currently active point mass are generated. (The number of event handlers in the tagger should be chosen accordingly). This general tagger can be used with any event handler realizing factors of Section 4.1 that is not connected to a cell system (the taggers suited for these event handlers are treated below).

- `ActiveRootUnitInStateTagger`:
  Generates the identifier of the root cnode of every branch in the extracted global state when the in-state identifiers are requested. This is used by event handlers that require the tree of an entire composite point object, which is either independent or induced active, to compute the candidate event time. An example for such an event handler is the mode switching event handler of Section 4.2.5.

- `CellBoundingPotentialTagger`:
  Uses a cell-occupancy system to generate the in-state identifiers. Iterates over all identifiers of active units and their active cells that are stored in the cell-occupancy system via the `yield_active_cells` method. It then generates the pairs of identifiers consisting of the respective active unit's identifier, and one by one the identifiers in non-excluded cells with respect to the active cell. The number of event handlers in the tagger should be chosen accordingly. This tagger is used with the event handlers of Sections 4.1.3 and 4.1.5 that use cell-based bounding potentials (see Section 4.1.1.6).

- `CellVetoTagger`:
  Uses a cell-occupancy system to generate the in-state identifiers. Here, it generates all active unit identifiers that are stored in the internal state. The number of event handlers in the tagger should be chosen accordingly. This tagger is used with the event handlers of Sections 4.1.4 and 4.1.5 that implement the computation of cell-veto events.

- `ExcludedCellsTagger`:
  Uses a cell-occupancy system to generate the in-state identifiers. Iterates over all identifiers of active units and their active cells that are stored in the cell-occupancy system via the `yield_active_cells` method. It then generates the pairs of identifiers consisting of the respective active unit's identifier, and one by one the identifiers in excluded cells with respect to the active cell. The number of event handlers in the tagger should be chosen accordingly. The event handlers used in this tagger must therefore be able to handle in-states that consist of branches of two of the stored identifiers in the cell-occupancy system. This tagger allows, for example, to generate in-state identifiers where a cell-based bounding potential would diverge.

- `SurplusCellsTagger`:
  Uses a cell-occupancy system to generate the in-state identifiers. Iterates over all identifiers of active units that are stored in the cell-occupancy system via the `yield_active_cells` method. It then generates the pairs of identifiers consisting of the respective active unit's identifier, and one by one the surplus identifiers generated by the `yield_surplus` method. The number of event handlers in the tagger should be chosen accordingly. The event handlers used in this tagger must therefore be able to handle in-states that consist of branches of two of the stored identifiers in the cell-occupancy system. This tagger allows, for example, to generate in-state identifiers that correspond to the surplus factors that are not included in the set of factors treated by a cell-veto event.

- `CellBoundaryTagger`: Uses a cell-occupancy system to generate the in-state identifiers. For this, it generates all active unit identifiers that are stored in the internal state. This tagger is used with the cell-boundary event handler of Section 4.2.1 whose events are required to keep the internal state consistent with the global state.

APPENDIX B

Publication

Philipp Höllmer, Liang Qin, Michael F. Faulkner, A. C. Maggs, Werner Krauth
JELLYFYSH-Version1.0 – a Python application for all-atom event-chain Monte Carlo
arXiv: 1907.12502 [physics.comp-ph] (2019)

This article presents JELLYFYSH-Version1.0, the general-purpose Python application for event-chain Monte Carlo (ECMC) that was introduced in detail in this thesis (see Chapters 3–6). It introduces the architecture of the application that closely mirrors the mathematical formulation of ECMC. Moreover, a number of worked-out examples for systems of atoms, dipoles, or water molecules are discussed.

At the submission date of this thesis, this article was in review for publication.

JeLLyFysh-Version1.0 - a Python application for all-atom event-chain Monte Carlo

Philipp Höllmer[a,b], Liang Qin[a], Michael F. Faulkner[c], A. C. Maggs[d], Werner Krauth[a,e,*]

[a] *Laboratoire de Physique de l'Ecole normale supérieure, ENS, Université PSL, CNRS, Sorbonne Université, Université Paris-Diderot, Sorbonne Paris Cité, Paris, France*
[b] *Bethe Center for Theoretical Physics, University of Bonn, Nussallee 12, 53115 Bonn, Germany*
[c] *H. H. Wills Physics Laboratory, University of Bristol, Tyndall Avenue, Bristol BS8 1TL, United Kingdom*
[d] *CNRS UMR7083, ESPCI Paris, PSL Research University, 10 rue Vauquelin, 75005 Paris, France*
[e] *Max-Planck-Institut für Physik komplexer Systeme, Nöthnitzer Str. 38, 01187 Dresden, Germany*

Abstract

We present JeLLyFysh-Version1.0, an open-source Python application for event-chain Monte Carlo (ECMC), an event-driven irreversible Markov-chain Monte Carlo algorithm for classical N-body simulations in statistical mechanics, biophysics and electrochemistry. The application's architecture closely mirrors the mathematical formulation of ECMC. Local potentials, long-ranged Coulomb interactions and multi-body bending potentials are covered, as well as bounding potentials and cell systems including the cell-veto algorithm. Configuration files illustrate a number of specific implementations for interacting atoms, dipoles, and water molecules.

1. Introduction

Event-chain Monte Carlo (ECMC) is an irreversible continuous-time Markov-chain algorithm [5, 28] that often equilibrates faster than its reversible counterparts [30, 19, 22, 23, 24]. ECMC has been successfully applied to the classic N-body all-atom problem in statistical physics [4, 17]. The algorithm implements the time evolution of a piecewise non-interacting, deterministic, system [6]. Each straight-line, non-interacting leg of this time evolution terminates in an event, defined through the event time at which it takes place and through the out-state, the updated starting configuration for the ensuing leg. An event is chosen as the earliest of a set of candidate events, each of which is sampled using information contained in a so-called factor. The entire trajectory samples the equilibrium probability distribution.

*Corresponding author, email address: werner.krauth@ens.fr

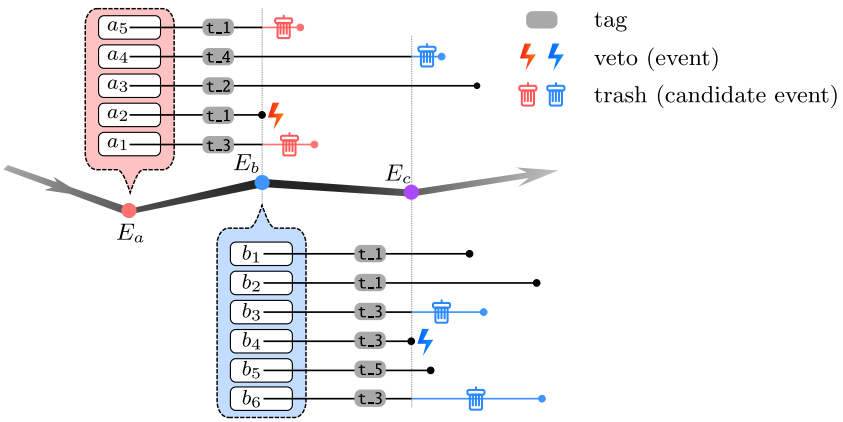


Figure 1: ECMC time evolution. At events $E_a, E_b, E_c, \ldots$, a number of factors $(\{a_1, a_2, \ldots, a_5\}, \{b_1, b_2, \ldots, b_6\}, \ldots)$ are activated. For each leg $((E_a \to E_b)$, $(E_b \to E_c)$, $\ldots)$, each factor must at all times independently accept the continued non-interacting evolution, and must determine a candidate event time at which this is no longer the case. The earliest candidate event time (which determines the veto) and its out-state yield the next event (the event E_b is triggered by a_2). In JF-V1.0, after committing an event to the global state, candidate events with certain tags are trashed (tags `t_1`, `t_3` at E_b) or maintained active (tags `t_2`, `t_4` at E_b), and others are newly activated. JF introduces non-confirmed events and also pseudo-factors, which complement the factors of ECMC, and which may also trigger events.

ECMC departs from virtually all Monte Carlo methods in that it does not evaluate the equilibrium probability density (or its ratios). In statistical physics, ECMC thus computes neither the total potential (or its changes) nor the total force on individual point masses. Rather, the decision to continue on the current leg of the non-interacting time evolution builds on a consensus which is established through the factorized Metropolis algorithm [28]. A veto puts an end to the consensus, triggers the event, and terminates the leg (see Fig. 1). In the continuous problems for which ECMC has been conceived, the veto is caused by a single factor.

The resulting event-driven ECMC algorithm is reminiscent of molecular dynamics, and in particular of event-driven molecular dynamics [1, 2, 3], in that there are velocity vectors (which appear as lifting variables). These velocities do not correspond to the physical (Newtonian) dynamics of the system. ECMC differs from molecular dynamics in three respects: First, ECMC is event-driven, and it remains approximation-free, for any interaction potential [32], whereas event-driven molecular dynamics is restricted to hard-sphere or piecewise constant potentials. (Interaction potentials in biophysical simulation codes have been coarsely discretized [8] in order to fit into the event-driven framework [36, 33, 34].) Second, in ECMC, most point masses are at rest at any time, whereas in molecular dynamics, all point masses typically have non-zero velocities. In ECMC, an arbitrary fixed number of (independently) active

point masses (with non-zero velocities) and identical velocity vectors for all of them may be chosen. In JF-V1.0, as in most previous applications of ECMC, only a single independent point mass is active. The ECMC dynamics is thus very simple, yet it mixes and relaxes at a rate at least as fast as in molecular dynamics [19, 23, 24]. Third, ECMC by construction exactly samples the Boltzmann (canonical) distribution, whereas molecular dynamics is in principle micro-canonical, that is, energy-conserving. Molecular dynamics is thus generally coupled to a thermostat in order to yield the Boltzmann distribution. The thermostat there also eliminates drift in physical observables due to integration errors. ECMC is free from truncation and discretization errors.

ECMC samples the equilibrium Boltzmann distribution without being itself in equilibrium, as it violates the detailed-balance condition. Remarkably, it establishes the aforementioned consensus and proceeds from one event to the next with $\mathcal{O}(1)$ computational effort even for long-range potentials, as was demonstrated for soft-sphere models, the Coulomb plasma [18, 19], and for the simple point-charge with flexible water molecules (SPC/Fw) model [39, 9].

JeLLyFysh (JF) is a general-purpose Python application that implements ECMC for a wide range of physical systems, from point masses interacting with central potentials to composite point objects such as finite-size dipoles, water molecules, and eventually peptides and polymers. The application's architecture closely mirrors the mathematical formulation that was presented previously (see [9, Sect II]). The application can run on virtually any computer, but it also allows for multiprocessing and, in the future, for parallel implementations. It is being developed as an open-source project on GitHub. Source code may be forked, modified, and then merged back into the project (see Section 6 for access information and licence issues). Contributions to the application are encouraged.

The present paper introduces the general architecture and the key features of JF. It accompanies the first public release of the application, JeLLyFysh-Version1.0 (JF-V1.0). JF-V1.0 implements ECMC for homogeneous, translation-invariant N-body systems in a regularly shaped periodic simulation box and with interactions that can be long-ranged. In addition, the present paper presents a cookbook that illustrates the application for simplified core examples that can be run from configuration files and validated against published data [9]. A full-scale simulation benchmark against the Lammps application is published elsewhere [35].

The JF application presented in this paper is intended to grow into a basis code that will foster the development of irreversible Markov-chain algorithms and will apply to a wide range of computational problems, from statistical physics to field theory [13]. It may prove useful in domains that have traditionally been reserved to molecular dynamics, and in particular in the all-atom Coulomb problem in biophysics and electrochemistry.

The content of the present paper is as follows: The remainder of Section 1 discusses the general setting of JF as it implements ECMC. Section 2 describes its mediator-based architecture [10]. Section 3 discusses how the eponymous events of ECMC are determined in the event handlers of JF. Section 4 presents

system definitions and tools, such as the user interface realized through configuration files, the simulation box, the cell systems, and the interaction potentials. Section 5, the cookbook, discusses a number of worked-out examples for previously presented systems of atoms, dipoles or water molecules with Coulomb interactions [9]. Section 6 discusses licence issues, code availability and code specifications. Section 7 presents an outlook on essential challenges and a preview of future releases of the application.

1.1. Configurations, factors, pseudo-factors, events, event handlers

In ECMC, configurations $c = \{\mathbf{s}_1, \ldots, \mathbf{s}_i, \ldots, \mathbf{s}_N\}$ are described by continuous time-dependent variables where $\mathbf{s}_i(t)$ represents the position of the ith of N point masses (although it may also stand for the continuous angle of a spin on a lattice [30]). JF is an event-driven implementation of ECMC, and it treats point masses and certain collective variables (such as the barycenter of a composite point object) on an equal footing. Rather than the time-dependent variables $\mathbf{s}_i(t)$, its fundamental particles (`Particle` objects) are individually time-sliced positions (of the point masses or composite point objects). Non-zero velocities and time stamps are also recorded, when applicable. The full information can be packed into units (`Unit` objects), that are moved around the application (see Section 1.2).

Each configuration c has a total potential $U(\{\mathbf{s}_1, \ldots, \mathbf{s}_N\})$, and its equilibrium probability density π is given by the Boltzmann weight

$$\pi(\{\mathbf{s}_1, \ldots, \mathbf{s}_N\}) = \exp\left[-\beta U(\{\mathbf{s}_1, \ldots, \mathbf{s}_N\})\right], \tag{1}$$

that is sampled by ECMC (see [9]). The total potential U is decomposed as

$$U(\{\mathbf{s}_1, \ldots, \mathbf{s}_N\}) = \sum_{M \in \mathcal{M}} U_M(\{\mathbf{s}_i : i \in I_M\}), \tag{2}$$

and the Boltzmann weight of eq. (1) is written as a product over terms that depend on factors M, with their corresponding factor potentials U_M. A factor $M = (I_M, T_M)$ consists of an index set I_M and of a factor type T_M, and $\mathcal{M}$ is the set of factors that have a non-zero contribution to eq. (2) for some configuration c. In the SPC/Fw water model, for example, one factor M with factor type T_M = Coulomb might describe all the Coulomb potentials between two given water molecules, and the factor index set I_M would contain the identifiers (indices) of the involved four hydrogens and two oxygens (see Section 5.3).

ECMC relies on the factorized Metropolis algorithm [28], where the move from a configuration c to another one, c', is accepted with probability

$$p^{\text{Fact}}(c \to c') = \prod_M \min\left[1, \exp\left(-\beta \Delta U_M\right)\right], \tag{3}$$

where $\Delta U_M = U_M(c'_M) - U_M(c_M)$. Rather than to evaluate the right-hand side of eq. (3), the product over the factors is interpreted as corresponding to a conjunction of independent Boolean random variables

$$X^{\text{Fact}}(c \to c') = \bigwedge_{M \in \mathcal{M}} X_M(c_M \to c'_M). \tag{4}$$

In this equation, $X^{\mathrm{Fact}}(c \to c')$ is "True" (the proposed Monte Carlo move is accepted) if the independently sampled factorwise Booleans X_M are all "True". Equivalently, the move $c \to c'$ is accepted if it is independently accepted by all factors. This realizes the aforementioned consensus decision (see Fig. 1). For an infinitesimal displacement, the random variable X_M of only a single factor M can be "False", and the factor M vetoes the consensus, creates an event, and starts a new leg. In this process, M requires only the knowledge of the factor in-state (based on the configuration c_M, and the information on the move), and the factor out-state (based on c'_M) provides all information on the evolution of the system after the event. The event is needed in order to enforce the global-balance condition (see Fig. 2a). In this process, lifting variables [7], corresponding to generalized velocities, allow one to repeat moves of the same type (same particle, same displacement), as long as they are accepted by consensus.[1] Physical and lifting variables build the overcomplete description of the Boltzmann distribution at the base of ECMC, and they correspond to the global physical and global lifting states of JF, its global state.

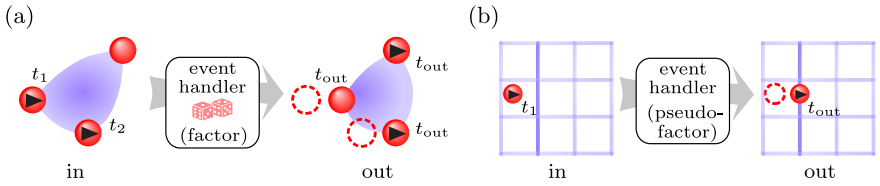


Figure 2: Factors and pseudo-factors. (a): In-state and sampled out-state (each with two active units) for a three-unit factor M (implementing, for example, the inter-molecular bending potential U_M of Section 4.4.5). (b): In- and out-states for a cell-boundary event handler realizing a pseudo-factor. Times at which units are time-sliced are indicated. t_{out} is the event time.

JF, the computer application, is entirely formulated in terms of events, beyond the requirements of the implemented event-driven ECMC algorithm. The application relies on the concept of pseudo-factors, which complement the factors in eq. (2), but are independent of potentials and without incidence on the global-balance condition (see Fig. 2b). In JF, the sampling of configuration space, for example, is expressed through events triggered by pseudo-factors. Pseudo-factors also trigger events that interrupt one continuous motion (one "event chain" [5]) and start a new one. Even the start and the end of each run of the application are formulated as events triggered by pseudo-factors.

In ECMC, among all factors M in eq. (2), only those for which U_M changes along one leg can trigger events. In JF, these factors are identified in a separate element of the application, the activator (see Section 2.4), and they are realized

[1] For concreteness, the lifting variables in this paper are referred to as "velocities", although they are not derived from mechanical equations of motions and their conservation laws. The concept of lifting variables is more general [7].

in yet other elements, the event handlers. An event handlers may require an in-state. It then computes the candidate event time and its out-state (from the in-state, from the factor potential, and from random elements). The complex operation of the activator and the event handlers is organized in JF-V1.0 with the help of a tag activator, with tags essentially providing finer distinction than the factor types T_M. A tagger identifies a certain pool of factors, and also singles out factors that are to be activated for each tag. The triggering of an event associated with a given tag entails the trashing of candidate events with certain tags, while other candidate events are maintained (see Fig. 1). Also, new candidate events have to be computed by event handlers with given tags. This entire process is managed by the tag activator.

1.2. Global state, internal state

In the event-driven formulation of ECMC, a point mass with identifier σ and with zero velocity is simply represented through its position, while an active point mass (with non-zero velocity) is represented through a time-sliced position $\mathbf{s}_\sigma(t_\sigma)$, a time stamp $\sigma(t_\sigma)$ and a velocity $\mathbf{v}_\sigma$:

$$\mathbf{s}_\sigma(t) = \begin{cases} \mathbf{s}_\sigma & \text{if } \mathbf{v}_\sigma = 0 \\ \mathbf{s}_\sigma(t_\sigma) + (t - t_\sigma)\mathbf{v}_\sigma & \text{else (active point mass)} \end{cases} . \tag{5}$$

An active point mass thus requires storing of a velocity $\mathbf{v}_i$ and of a time stamp t_σ, in addition to the time-sliced position $\mathbf{s}_\sigma(t_\sigma)$. In JF, the global state traces all the information in eq. (5). It is broken up into the global physical state, for the time-sliced positions $\mathbf{s}_\sigma$, and the global lifting state, for the non-zero velocities $\mathbf{v}_\sigma$ and the time stamps t_σ.

JF represents composite point objects as trees described by nodes. Leaf nodes correspond to the individual point masses. A tree's inner nodes may represent, for example, the barycenters of parts of a molecule, and the root node that of the entire molecule (see Fig. 3a-b). The velocities inside a composite point object are kept consistent, which means that the global lifting state includes non-zero velocities and time stamps of inner and root nodes. The storing element of the global state in JF is the state handler (see Section 2.3). The global state is not directly accessed by other elements of the application, but branches of the tree can be extracted (copied) temporarily, together with their unit information. Independent and induced units differentiate between those that appear in ECMC and those that are carried along in order to assure consistency (see Fig. 3).

For internal computations, the global state may be supplemented by an internal state that is kept, not in the state handler, but in the activator part of the application (see Section 2.4). In JF-V1.0, the internal state consists in cell-occupancy systems, which associate identifiers of composite point objects or point masses to cells. (An identifier is a generalized particle index with, in the case of a tree, a number of elements that correspond to the level of the corresponding node.) In JF, cell-occupancy systems are used for book-keeping,

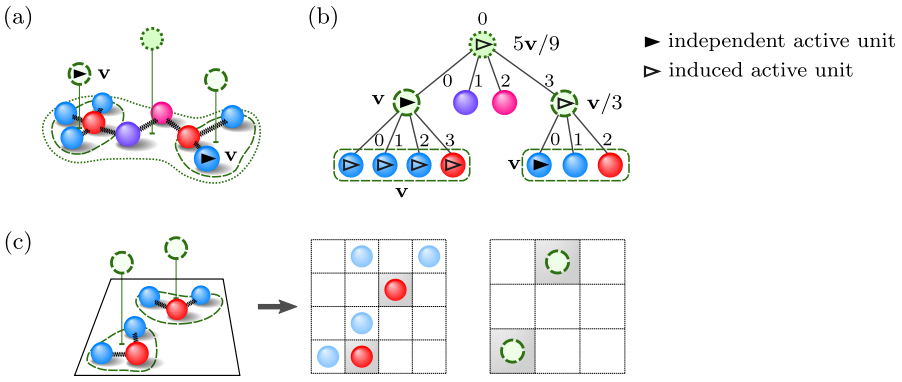


Figure 3: Tree representation of composite point objects in JF-V1.0. (a): Molecule with functional parts. (b): Tree representation, with leaf nodes for the individual atoms and higher-level nodes for barycenters. Nodes each have a particle (a `Particle` object) containing a position vector and charge values. A unit (a `Unit` object), associated with a node, copies out the particle's identifier and its complete global-state information. (c): Internal representation of composite point objects with separate cell systems for particle identifiers on different levels. On the leaf level, only one kind of particles is tracked.

and also for cell-based bounding potentials. JF-V1.0 requires consistency between the time-sliced particle information and the units. This means that the time-sliced position $\mathbf{s}_\sigma(t_\sigma)$ and the time-dependent position $\mathbf{s}_\sigma(t)$ in eq. (5) belong to the same cell (see Fig. 2b). Several cell-occupancy systems may coexist within the internal state (possibly on different tree-levels and with different cell systems, see Fig. 3c and Section 5.3.4). ECMC requires time-slicing only for units whose velocities are modified. Beyond the consistency requirements, JF-V1.0 performs time-slicing also for unconfirmed events, that is, for triggered events for which, after all, the out-state continues the straight-line motion of the in-state (see Section 3.1.2).

1.3. Lifting schemes

In its lifted representation of the Boltzmann distribution, ECMC introduces velocities for which there are many choices, that is, lifting schemes. The number of independent active units can in particular be set to any value $n_{\mathrm{ac}} > 1$ and then held fixed throughout a given run. This generalizes easily from the known $n_{\mathrm{ac}} = 1$ case [12]. A simple n_{ac}-conserving lifting scheme uses a factor-derivative table (see [9, Fig. 2]), but confirms the active out-state unit only if the corresponding unit is not active in the in-state (its velocity is `None`). For $|I_M| > 3$, the lifting scheme (the way of determining the out-state given the in-state) is not unique, and its choice influences the ECMC dynamics [9]. In JF-V1.0, different lifting-scheme classes are provided in the JF `lifting` package. They all construct independent-unit out-state velocities for independent units that equal the in-state velocities. This appears as the most natural choice in spatially homogeneous systems [5].

1.4. Multiprocessing

In ECMC, factors are statistically independent. In JF, therefore, the event handlers that realize these factors can be run independently on a multiprocessor machine. With multiprocessor support enabled, candidate events are concurrently determined by event handlers on separate processes, using the Python `multiprocessing` module. Candidate event times are then first requested in parallel from active event handlers, and then the out-state for the selected event. Given a sufficient number of available processors, out-states may be computed for candidate events in advance, before they are requested (see Section 2.1). The event handlers themselves correspond to processes that usually last for the entire duration of one ECMC run. When not computing, event handlers are either in `idle` stage waiting to compute an candidate event time or in `suspended` stage waiting to compute an out-state.

Using multiple processes instead of threads circumvents the Python global interpreter lock, but the incompressible time lag due to data exchange slows down the multiprocessor implementation of the mediator with respect to the single-processor implementation.

1.5. Parallelization

ECMC generalizes to more than one independent active unit, and a sequential, single-process ECMC computation remains trivially correct for arbitrary n_{ac} (although JF-V1.0 only fully implements the $n_{\mathrm{ac}} = 1$ case). The relative independence of a small number of independent active units in a large system, for $1 \ll n_{\mathrm{ac}} \ll N$, allows one to consider the simultaneous committing in different processes of n_{pr} events to the shared global state. (A conflict arises if this disagrees with what would result by committing them in a single process.) If $n_{\mathrm{pr}} \ll N$, conflicts between processes disappear (for short-range interacting systems) if nearby active units are treated in a single process (see Fig. 4a). The parallel implementation of ECMC, for short-range interactions, is conceptually much simpler than that of event-driven molecular dynamics [29, 14, 20], and it may well extend to long-range interacting system.

An alternative type of parallel ECMC, domain decomposition into n_{ac} stripes, was demonstrated for two-dimensional hard-spheres systems, and considerable speed-up was reached [16]. Here, stripes are oriented parallel to the velocities, with one active unit per stripe. Stripes are isolated from each other by immobile layers of spheres [16], which however cause rejections (or reversals of one or more components of the velocity). The stripe decomposition eliminates all scheduling conflicts. As any domain decomposition [29], it is restricted to physical models with short-range interactions. It is not implemented in JF-V1.0 (see Fig. 4b).

2. JF architecture

JF adopts the design pattern based on a mediator [10], which serves as the central hub for the other elements that do not directly connect to each other. In this way, interfaces and data exchange are particularly simple. The mediator design maximizes modularity in view of future extensions of the application.

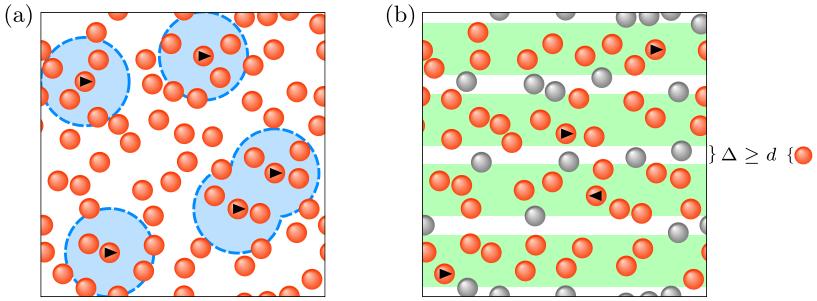


Figure 4: Parallel ECMC with local potentials (interaction range d). (a): Multiprocess version with $n_{\rm ac} \ll N$ active units. Nearby active units avoid conflict in a single process. (b): Domain decomposition with separated stripes. Particles in between stripes are immobile. The separation region (of width Δ) is wider than d, so that all conflict between stripes is avoided (see [16]).

2.1. Mediator

The mediator is doubled up into two modules (with `SingleProcessMediator` and `MultiProcessMediator` classes). The `run` method of either class is called by the executable `run.py` script of the application, and it loops over the legs of the continuous-time evolution. The loop is interrupted when an `EndOfRun` exception is raised, and a `post_run` method is invoked. For the single-process mediator, all the other elements are instances of classes that provide public methods. In particular, the mediator interacts with event handlers. For the multi-process mediator, each event handler has its own autonomous iteration loop and runs in a separate process. It exchanges data with the mediator through a two-way pipe. Receiving ends on both sides detect when data is available using the pipe's `recv` methods.

In JF-V1.0, the same event-handler classes are used for the single-process and multi-process mediator classes. The multi-process mediator achieves this through a monkey-patching technique. It dynamically adds a `run_in_process` method to each created instance of an event handler, which then runs as an autonomous iteration loop in a process and reacts to shared flags set by the mediator. The multi-process mediator in addition decorates the event handler's `send_event_time` and `send_out_state` methods so that output is not simply returned (as it is in the single-process mediator) but rather transmitted through a pipe. Only the mediator accesses the event handlers, and these re-definitions of methods and classes (which abolish the need for two versions for each event-handler class) are certain not to produce undesired side effects.

On one leg of the continuous-time evolution, the mediator goes through nine steps (see Fig. 5). In step 1, the active global state (that part of the global state that appears in the global lifting state) is obtained from the state handler. (In the tree state handler of JF-V1.0, branches of independent units are created

for all identifiers that appear in the lifting state.) Knowing the preceding event handler (which initially is `None`) and the active global state, it then obtains from the activator, in step 2, the event handlers to activate together with their in-state identifiers. For this, the activator may rely on its internal state, but not on the global state, to which it has no access. In step 3, the corresponding in-states are extracted (that is, copied) from the state handler. In step 4, candidate event times are requested from the appropriate event handlers and pushed into the scheduler's `push_event` method. In step 5, the mediator obtains the earliest candidate event time from the scheduler's `get_succeeding_event` method and and asks its event handler for the event out-state (step 6) to be committed to the global state (step 7). The activator, in step 8, determines which candidate events are to be trashed (in JF-V1.0: based on their tags), that is, which candidate event times are to be eliminated from the scheduler. Also, the activator collects the corresponding event handlers, as they become available to determine new candidate events. In the optional final step 9, the mediator may connect (*via* the input–output handler) to an output handler, depending on the preceding event handler. A mediating method defines the arguments sent to the output handler (for example the extracted global state), and considerable computations may take place there.

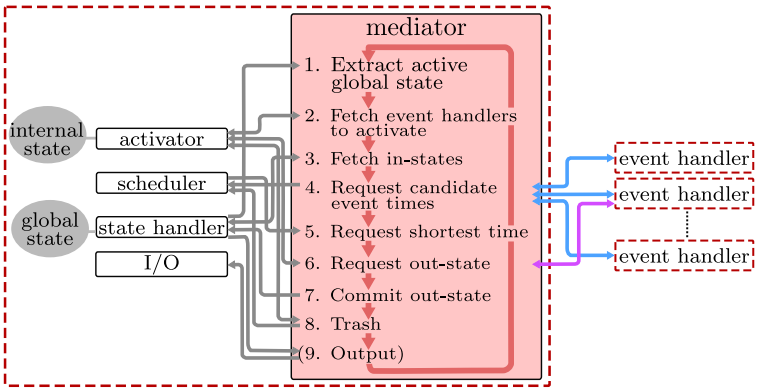


Figure 5: JF architecture, built on the mediator design pattern. The iteration loop takes the system from one event to the next (for example from E_a to E_b in Fig. 1). All elements of JF interact with the mediator, but not with each other. The multi-process mediator interacts with event handlers running on separate processes, and exchanges data *via* pipes.

The multi-process mediator uses a single pipe to receive the candidate event time and the out-state from an event handler. In order to distinguish the received object, the mediator assigns four different stages to the event handlers (`idle`, `event_time_started`, `suspended`, `out_state_started` stages). The assigned stage determines which flags can be set to start the `send_event_time` or `send_out_state` methods. It also determines the nature of the data contained in the pipe. In the `idle` stage, the mediator can set the starting flag after which the event handler will wait to receive the in-state through the pipe. This starts

the `event_time_started` stage during which the event handler determines the next candidate event time and places it into the pipe. After the mediator has recovered the data from the pipe, it places the event handler into the `suspended` stage. If requested (by flags), the event handler can then either compute the out-state (`out_state_started` stage), or else revert to the `event_time_started` stage.

The strategy for suspending an event handler or for having it start an out-state computation (before the request) can be adjusted to the availability of physical processors on the multi-processor machine. However, in JF-V1.0, the communication *via* pipes presents a computational bottleneck.

2.2. Event handlers

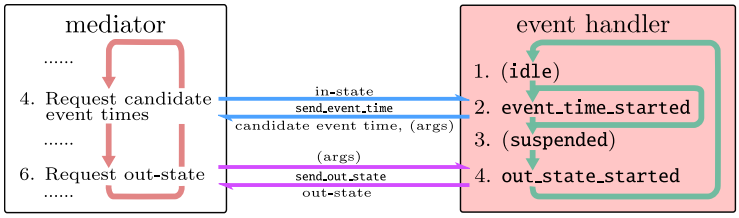


Figure 6: Basic stages of event handlers for factors and pseudo-factors (stages 1 and 3 relevant for the multi-process mediator only). In the `idle` and `suspended` stages, the event handler is halted (*via* flags controlled by the multi-process mediator), thus liberating resources for other candidate-event-time requests. With the multi-process mediator, candidate out-states may be computed before the out-state request arrives.

Event handlers (instances of a number of classes that inherit from the abstract `EventHandler` class) provide the `send_event_time` and `send_out_state` methods that return candidate events. These candidate events either become events of a factor or pseudo-factor or they will be be trashed.[2]

When realizing a factor or a pseudo-factor, event handlers receive the in-state as an argument of the `send_event_time` method. The `send_out_state` method then takes no argument. In contrast, event handlers that realize a set of factors or pseudo-factors request candidate event times without first specifying the complete in-state, because the element of the set that triggers the event is yet unknown at the event-time request (see Section 3.2.2 for examples of event handlers that realize sets of factors). The `send_event_time` method then takes the part of the in-state which is necessary to calculate the candidate event time. Also, it may return supplementary arguments together with the candidate event time, which is used by the mediator to construct the full in-state. The in-state is then an argument of the `send_out_state` method, as it was not sent earlier.

[2]A candidate event time may stem from a bounding potential, and not be confirmed for the factor potential. In JF-V1.0, unconfirmed and confirmed events are treated alike.

In JF-V1.0, each run requires a start-of-run event handler (an instance of a class that inherits from the abstract `StartOfRunEventHandler` class), and it cannot terminate properly without an end-of-run event handler. Section 3 discusses several event-handler classes that are provided.

2.3. State handler

The state handler (an instance of a class that inherits from the abstract `StateHandler` class) is the sole separate element of JF to access the global state. In JF-V1.0, the global physical state (all positions of point masses and composite point objects) is contained in an instance of the `TreePhysicalState` class represented as a tree consisting of nodes (each node corresponds to a `Node` object). Each node contains a particle (a `Particle` object) which holds a time-sliced position. In JF-V1.0, each leaf node may in addition have charges as a Python dictionary mapping the name of the charge onto its value.

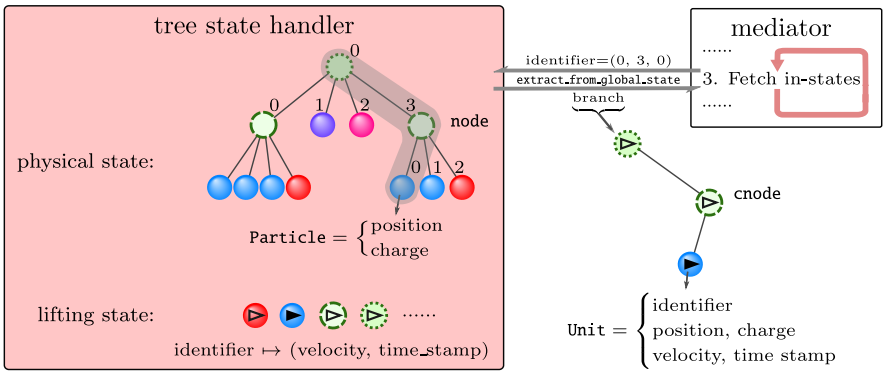


Figure 7: Inner storage of the tree state handler and example of its `extract_from_global_state` method, applied to the global state of Fig. 3b.

Each tree is specified through its root node. Root nodes can be iterated over (in JF-V1.0, they are members of a list). Each node is connected to its parent and its children, which can also be iterated over. In JF-V1.0, the children are again members of a list. These lists imply unique identifiers of nodes and their particles as tuples. The first entry of the tuple gives a node's root node list index, followed by the indices on lower levels down to the node itself (see Fig. 3).

The global lifting state is stored in JF-V1.0 in a Python dictionary mapping the implicit particle identifier onto its time stamp and its velocity vector. This information is contained in an instance of the `TreeLiftingState` class. Both the physical and lifting states are combined in the `TreeStateHandler` which implements all methods of a state handler.

To communicate with other elements of the JF application (such as the event handlers and the activator) *via* the mediator, the state handler combines the

information of the global physical and the global lifting state into units (that is, temporary `Unit` objects, see Fig. 7). A given physical-state and lifting-state information for a node in the state handler is mirrored (that is copied) to a unit containing its implicit identifier, position, charge, velocity and time stamp. All other elements can access, modify, and return units. This provides a common packaging format across JF. The explicit identifier of a unit allows the program to integrate changed units into the state handler's global state.

In the tree state handler of JF-V1.0, the local tree structure of nodes can be extracted into a branch of cnodes, that is, nodes containing units.[3] Each event handler only requires the global state reduced to a single factor in order to determine candidate event times and out-states. As a design principle in JF-V1.0, the event handlers keep the time-slicing of composite point objects and its point masses consistent. Information sent to event handlers *via* the mediator is therefore structured as branches, that is the information of a node with its ancestors and descendants. The state handler's `extract_from_global_state` method creates a branch for a given identifier of a particle by constructing a temporary copy of the immutable node structure of the state handler using cnodes. Out-states of events in the form of branches can be committed to the global state using the `insert_into_global_state` method.

The `extract_active_global_state` method, the first of two additional methods provided by the state handler, extracts the part of the global state which appears in the global lifting state. The tree state handler constructs the minimal number of branches, where each node contains an active unit, so that all implicit identifiers appearing in the global lifting state are represented. The activator may then determine the factors which are to be activated. The method is also used to time-slice the entire global state (see Section 3.2.2). Second, the `extract_global_state` method extracts the full global state. (For the tree state handler of JF-V1.0, this corresponds to a branch for each root node.) This method does not copy the positions and velocities.

In JF-V1.0, the global physical state is initialized *via* the input handler within the input–output handler (see Section 2.6). The initial lifting state, however, is set *via* the out-state of the start-of-run event handler, which is committed to the global state at the beginning of the program (see Section 3.2.2). This means that, in JF-V1.0, the lifting state cannot be initialized from a file.

2.4. Activator

The activator, a separate element of the JF application, is an instance of a class that inherits from the abstract `Activator` class. At the beginning of each leg, the activator provides to the mediator the new event handlers which are to be run, using the `get_event_handlers_to_run` method. (As required by the mediator design pattern, no data flows directly between the activator and the

[3]The distinction between particles and units, as well as between nodes and cnodes stresses that the state handler can only be accessed by the mediator, although information on the physical and the lifting state must of course travel throughout the application.

event handlers, although it initially obtains their references, and subsequently manages them.) The activator also returns associated in-state identifiers of particles within the global state. The extracted parts of the global state of these are needed by the event handlers to compute their candidate event time (the identifier may be `None` if no information is needed). Finally, it readies for the mediator a list of trashable candidate events at the end of each leg in the `get_trashable_events` method, once the mediator has committed the preceding event to the global state *via* the state handler.

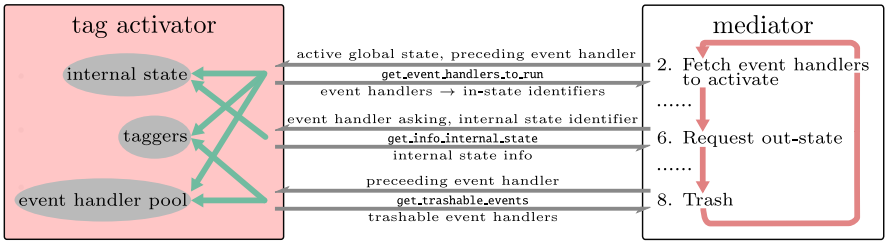


Figure 8: Tag activator, and its complex interaction with the mediator. It readies event handlers and in-state identifiers, provides internal-state information for an out-state request, and identifies the trashable candidate events, as a function of the preceding event.

In JF-V1.0, the activator is an instance of the `TagActivator` class (that inherits from the `Activator` class). The tag activator's operations depend on the interdependence of tags of event handlers and their events. Event handlers receive their tag by instances of classes located in the activator and derived from the abstract `Tagger` class that are called "taggers".

A tagger centralizes common operations for identically tagged event handlers (see Fig. 8). On initialization, the tagger receives its tag (a string-valued `tag` attribute) and an event handler (that is, a single instance), of which it creates as many identical event-handler copies as needed (using the Python `deepcopy` method). Each tagger provides a `yield_identifiers_send_event_time` method which generates in-state identifiers based on the branches containing independent active units (this means that the taggers are implemented especially for the `TreeStateHandler`, the `TagActivator` however is not restricted to this since it just transmits the extracted active global state). These in-states are passed (after extracting the part of the global state related to the identifiers from the state handler) to the `send_event_time` method of the tagger's event handlers. The number of event handlers inside a tagger should meet the maximum number of events with the given tag simultaneously in the scheduler. In this paper, event handlers (and their candidate events) are referred to by tags, although in JF they do not have the `tag` attribute of their taggers.

On initialization, a tagger also receives a list of tags for event handlers that it creates, as well as a list of tags for event handlers that need to be trashed. The tag activator converts this information of all taggers into its internal `_create_taggers` and `_trash_taggers` dictionaries. Additionally, the tag

activator creates an internal dictionary mapping from an event handler onto the corresponding tagger (`_event_handler_tagger_dictionary`).

A call of the `get_event_handlers_to_run` method is accompanied by the event handler which created the preceding event and by the extracted active global state. The event handler is first mapped onto its tagger. The taggers returned by the `_create_taggers` dictionary then generate the in-states identifiers, which are returned together with the corresponding event handlers (in a dictionary). For the initial call of the `get_event_handlers_to_run` method no information on the preceding event handler can be provided. This is solved by initially returning the start-of-run event handler. Similarly the `_trash_taggers` dictionary is used on each call of `get_trashable_events`. The corresponding event handlers are then also liberated, meaning that the activator can return them in the next call of the `get_event_handlers_to_run` method.[4] For this, the activator internally splits the pool of all event handlers of a given tag internally into those with a scheduled candidate event and the ones that are available to take on new candidate events.

The activator also maintains the internal state. In JF-V1.0, the internal state consists in cell-occupancy systems. Therefore, the internal state is an instance of a class that inherits from the `CellOccupancy` class, which itself inherits from the abstract `InternalState` class. Taggers may refer to internal-state information to determine the in-states of their event handlers. The cell-occupancy system does not double up on the information available in the state handler. It keeps track of the identifier of a particle (which may correspond to a point mass or a composite point object), but does not store or copy the particle itself (see Section 4.3). The mediator can access the internal state *via* the `get_info_internal_state` method (see Fig. 8). To acquire consistency between the global state and the internal state (and between a particle and its associated unit), a pseudo-factor triggers an event for each active unit tracked by the cell-occupancy system that crosses a cell boundary (see Fig. 2b). The internal state is updated in each call of the `get_event_handlers_to_run` method.

2.5. Scheduler

The scheduler is an instance of a class inheriting from the abstract `Scheduler` class. It keeps track of the candidate events and their associated event-handler references. Its `get_succeeding_event` method selects among the candidate events the one with the smallest candidate event time, and it returns the reference of the corresponding event handler. Its `push_event` method receives a new candidate event time and event-handler reference. Its `trash_event` method eliminates a candidate event, based on the reference of its event handler. In JF-V1.0, the scheduler is an instance of the `HeapScheduler` class. It implements a priority queue through the Python `heapq` module.

[4] The action of the `_create_taggers` and `_trash_traggers` dictionaries can be overruled with the concept of activated and deactivated taggers. Event handlers out of deactivated taggers are not returned to the mediator.

2.6. Input–output handler

The input–output handler is an instance of the `InputOutputHandler` class. The input–output handler connects the JF application to the outside world, and it is accessible by the mediator. The input–output handler breaks up into one input handler (an instance of a class that inherits from the abstract `InputHandler` class) and a possibly empty list of output handlers (instances of classes that inherit from the abstract `OutputHandler` class). These are accessed by the mediator only *via* the input–output handler. Output handlers can also perform significant calculations.

The input handler enters the initial global physical state into the application. JF-V1.0 provides an input handler that enters protein-data-bank formatted data (`.pdb` files) as well as an input handler which samples a random initial state. The initial state (constructed as a tree for the case of the tree state handler) is returned when calling the `read` method of the input–output handler, which calls the `read` method of the input handler.

The output handlers serve many purposes, from the output in `.pdb` files to the sampling of correlation functions and other observables, to a dump of the entire run. They obtain their arguments (for example the entire global state) *via* its `write` method. The `write` method of the input–output handler receives the desired output handler as an additional argument through the mediating methods of specific event handlers. These are triggered for example after a sampling or an end-of-run event. The corresponding event handlers are initialized with the name of their output handlers.

3. JF event-handler classes

Event handler classes differ in how they provide the `send_event_time` and `send_out_state` methods. Event handlers split into those that realize factors and sets of factors and those that realize pseudo-factors and sets of pseudo-factors. The first are required by ECMC while the second permit JF to represent the entire run in terms of events.

3.1. Event handlers for factors or sets of factors

Event handlers that realize a factor M, or a set of factors are implemented in different ways depending on the analytic properties of the factor potential U_M and on the number of independent active units.

3.1.1. Invertible-potential event handlers

In JF, an invertible factor potential U_M (an instance of a class that inherits from the abstract `InvertiblePotential` class) has its event rate integrated in closed form along a straight-line trajectory (see Fig. 1). The sampled cumulative event rate (U_M^+ in [9, eq. (45)]) provides the `displacement` method. Together with the time stamp and the velocity of the active unit, this determines the candidate event time. In JF-V1.0, the two-leaf-unit event handler (an instance of the `TwoLeafUnitEventHandler` class) is characterized by two

independent units at the leaf level. It realizes a two-particle factor with an invertible factor potential. The in-state (an argument of the `send_event_time` method) is stored internally, and it remains available for the subsequent call of the `send_out_state` method. Because of the two independent units, the lifting simply consists in these two units switching their velocities (using the internal `_exchange_velocity` method) and keeping the velocities of all induced units consistent.

3.1.2. Event handlers for factors with bounding potential

For a factor potential U_M that is not inverted (by choice or by necessity because it is non-invertible), the cumulative event rate U_M^+ is unavailable (or not used) and so is its `displacement` method. Only the `derivative` method is used. To realize such a factor without an inverted factor potential, an event handler then uses the `displacement` method of an associated bounding potential whose event rate at least equals that of U_M and that is itself invertible. A non-inverted U_M may be associated with more than one bounding potential, each corresponding to a different event handler (the molecular Coulomb factors in Section 5.2 associate the Coulomb factor potential in the same run with different bounding potentials). In JF-V1.0, a number of event handlers are instances of classes that inherit from the `EventHandlerWithBoundingPotential` class, and that realize factors with bounding potentials. Each of these event handlers translates the sampled displacement of the bounding potential into a candidate event time. On an out-state request (*via* the `send_out_state` method), the event handler confirms the event with probability that is given by the ratio of the event rates of the factor potential and the bounding potential. The out-state consists of independent units together with their branches of induced units. For two independent units, the lifting limits itself to the application of a local `_exchange_velocity` method, which exchanges independent-unit velocities and enforces velocities for the induced units. For more independent units, the out-state calculation requires a lifting. For an unconfirmed event, no lifting takes place. In JF-V1.0, confirmed and unconfirmed event have time-sliced out-states. Inefficient treatment of unconfirmed events is the main limitation of this version of the application.

A special case of a bounding potential is the cell-based bounding potential which features piecewise cell-bounded event rates. The two independent units are localized within their respective cells, and the bounding potential's rate is for all positions of the units larger than the factor potential event rate. In JF-V1.0, the constant cell-bounded event rate is determined for all pairs of cells on initialization (see Section 4.4.4). The resulting displacement may move the independent active unit outside its cell. The proposed candidate event will then however be preempted by a cell-boundary event and then trashed (see Section 3.2.1).

3.1.3. Cell-veto event handlers

Cell-veto event handlers (instances of a number of classes that inherit from the abstract `CellVetoEventHandler` class) realize sets of factors, rather than a

single factor. The factor in-states (for each element of the set) are not transmitted with the candidate-event-time request. Instead, the branch of the independent active unit is an argument of the `send_event_time` method. The sampled factor in-state is transmitted with the out-state request. The cell-veto event handler implements Walker's algorithm [37] in order to sample one element in the set of factors in $\mathcal{O}(1)$ operations.

Cell-veto event handlers are instantiated with an estimator (see Section 4.6). In addition, they obtain a cell system which is read in through its `initialize` method (see Section 4.2). The estimator provides upper limits for the event rate (in the given direction of motion) for the independent active unit anywhere in one specific cell (called "zero-cell", see Section 4.3), and for a target unit in any other cell, except for a list of excluded cells. These upper limits can be translated from the zero-cell to any other active-unit cell, because of the homogeneity of the simulation box. In JF-V1.0, the cell systems for the cell-veto event handler can be on any level of the particles' tree representation (see Section 5.3.4, where a molecule-cell system tracks individual water molecules on the root level, while a oxygen-cell system tracks only the leaf nodes corresponding to oxygens).

A Walker sampler is an instance of the `Walker` class in the `event_handler` package. It provides the total event rate (`total_rate`), that for a homogeneous periodic system is a constant throughout a run. On a candidate-event-time request, a cell-veto event handler computes a displacement, but no longer through the `displacement` method of a factor potential or a bounding potential, but simply as an exponential random number divided by the total event rate. (The particularly simple `send_event_time` method of a cell-veto event handler is implemented in the abstract `CellVetoEventHandler` class.) (see [9] for a full description). The Walker sampler's `sample_cell` method samples the cell of the target unit in $\mathcal{O}(1)$. It is returned, together with the candidate event time, as an argument of the `send_event_time` method. The out-state request is accompanied by the branch of the independent unit in the target cell, if it exists. Confirmation of events and, possibly, lifting are handled as in Section 3.1.2.

3.2. Event handlers for pseudo-factors or sets of pseudo-factors

The pseudo-factors of JF unify the description of the ECMC time evolution entirely in terms of events. The distinction between event handlers that realize pseudo-factors and those that realize sets of pseudo-factors remains crucial. In the former, the factor in-state is known at the candidate-event-time request. It is transmitted at this moment and kept in the memory of the event handler for use at the out-state request. For a set of pseudo-factors, the factor in-state can either not be specified at the candidate-event-time request, or would require transmitting too much data (one in-state per element of the set). It is therefore transmitted later, with the out-state-request (see Fig. 9).

3.2.1. Cell-boundary event handler

In the presence of a cell-occupancy system, JF-V1.0 preserves consistency between the tracked particles of the global physical state and the corresponding

units (which must belong to the same cell). This is enforced by a cell-boundary event handler, an instance of the `CellBoundaryEventHandler` class. This event handler has a single independent unit and realizes a pseudo-factor with a single identifier. A cell-boundary event leads to the internal state to be updated (see Section 2.4).

On instantiation, a cell-boundary event handler receives a cell system. (Each cell-occupancy system requires one independent cell-boundary event handler.) A candidate-event-time request by the mediator is accompanied by the in-state contained in a single branch and a single unit on the level tracked by the cell-occupancy system. An out-state request is met with the cell-level-unit's position corresponding to the minimal position in the new cell.

3.2.2. Event handlers for sampling, end-of-chain, start-of-run, end-of-run

Sampling event handlers are instances of classes that inherit from the abstract `SamplingEventHandler` class. Sampling event handlers are expected to produce output (they inherit from the `EventHandlerWithOutputHandler` class and are connected, on instantiation, with their own output handler which is used in the mediating method of this event handler). Several sampling event handlers may coexist in one run. Their output handler is responsible for computing physical observable at the sampling event time (see Section 2.6). JF-V1.0 implements sampling events as the time-slicing of all the active units. A sampling event handler thus realizes a set of single-unit pseudo-factors, and the in-state is not specified at the candidate-event-time request. In JF-V1.0, the candidate event times of the sampling event handler are equally spaced. The out-state request is accompanied by branches of all independent active units, which are then all time-sliced simultaneously. Sampling candidate events are normally trashed only by themselves and by an end-of-run event.

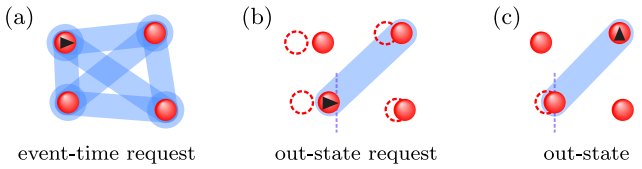


Figure 9: Set of pseudo-factors realized by the end-of-chain event handler. (a): Set of end-of-chain pair pseudo-factors for four point masses coupling the final active unit of the old chain and the beginning active unit of the new chain. (b): At the event time, the realized pseudo-factor with the incoming active unit and the outgoing unit is known. (c): A new event chain is started. The outgoing active unit is shown.

End-of chain event handlers are instances of classes that inherit from the abstract `EndOfChainEventHandler` class. They effectively stop one event chain and reinitialize a new one. This is often required for the entire run to be irreducible (see [9]). The end-of-chain event handler clearly realizes a set of pseudo-factors, rather than a single pseudo-factor (see Fig. 9a). An end-of-chain event handler implements a method to sample a new direction of motion.

In addition, it implements a method to determine a new chain length (that gives the time of the next end-of-chain event) and, finally, the identifiers of the next independent active cnodes. For this, the end-of-chain event handler is aware of all the possible cnode identifiers (see Section 4.2).

On an event-time request, the end-of-chain event handler returns the next candidate event time (computed from the new chain length) and the identifier of the next independent active cnode. The out-state request is accompanied by the current and the succeeding independent active units and their associated branches (see Fig. 9b). For the out-state, the event handler determines the next direction of motion (see Fig. 9c).

A start-of-run event handler (an instance of a class that inherits from the abstract `StartOfRunEventHandler` class) is the sole event handler whose presence is required. The start-of-run event is the first one to be committed to the global state, because its candidate event time is set equal to the initial time of the run (usually zero) and because the activator will initially only activate the start-of-run event handler. The start-of-run event handler serves two purposes. First, it sets the initial lifting state. Second, the activator uses the start-of-run event handler as an entry point. Its tag (the `start_of_run` tag in the configuration files of Section 5) is then used to determine the events that should be activated and created thereafter.

The end-of-run event handler (an instance of a class that inherits from the abstract `EndOfRunEventHandler` class) terminates a run by raising an end-of-run exception and thus ends the mediator loop. An end-of-run event handler is usually connected, on instantiation, with its own output handler. In JF-V1.0, its `send_event_time` method returns the total run-time, which transits from the configuration file. On the `send_out_state` request, all active units are time-sliced. The end-of-run output handler may further process the global state which it receives *via* the mediating method of the end-of-run event handler.

3.3. Event handlers for rigid motion of composite point objects, mode switching

The event handlers of JF-V1.0 are generally suited for the rigid motion of composite point objects (root mode), that is, for independent non-leaf-node units (as implemented in Section 5.2.4). This is possible because all event handlers keep the branches of independent units consistent. As the subtree-node units of an independent-unit node move rigidly, the displacement is not irreducible. Mode switching into leaf mode (with single active leaf units) then becomes a necessity in order to have all factors be considered during one run and to assure the irreversibility of the implemented algorithm. In JF-V1.0, the corresponding event handlers are instances of the `RootLeafUnitActiveSwitcher` class. On instantiation, they are specified to switch either from leaf mode to root mode or vice versa.

These event handlers resemble the end-of-chain event handler, but only one of them is active at any given time. They provide a method to sample the new candidate event time based on the time stamp of the active independent unit at the time of its activation. An out-state request from one of these event handlers is accompanied by the entire tree of the current independent active unit of one

mode and met with the tree of the independent active unit on the alternate mode.

4. JF run specifications and tools

The JF application relies on a user interface to select the physical system that is considered, and to fully specify the algorithm used to simulate it. Inside the application, some of these choices are made available to all modules (rather than having to be communicated repeatedly by the mediator). The application also relies on a number of tools that provide key features to many of its parts.

4.1. Configuration files, logging

The user interface for each run of the JF application consists in a configuration file that is an argument of the executable `run.py` script.[5] It specifies the physical and algorithmic parameters (temperature, system shape and size, dimension, type of point masses and composite point objects, and also factors, factor potentials, lifting schemes, total run time, sampling frequency, etc).

A configuration file is composed of sections that each correspond to a class requiring input parameters. The `[Run]` section specifies the mediator and the setting. The ensuing sections choose the parameters in the `__init__` methods of the mediator and of the setting. Each section contains pairs of properties and values. The property corresponds to the name of the argument in the `__init__` method of the given class, and its value provides the argument (see Fig. 12). The content of the configuration file is parsed by the `configparser` module and passed to the JF factory (located in the `base.factory` module) in `run.py`. Standard Python naming conventions are respected in the classes built by the JF factory, which implies the naming conventions in the configuration file (see Section 6.3 for details). Within the configuration file, sections can be written in any order, but their explicit nesting is not allowed. The nestedness is however implicit in the structure of the configuration file.

The JF application returns all output *via* files under the control of output handlers. Run-time information is logged (the Python `logging` module is used). Logged information can range from identification of CPUs to the initialization information of classes, run-time information, etc. Logging output (to standard output or to a file) can take place on a variety of levels from `DEBUG` to `INFO` to `WARNING` that are controlled through arguments of `run.py`. An identification hash of the run is part of the logging output. It also tags all the output files so that input, output and log files are uniquely linked (the Python `uuid` module is used).

[5] Configuration files follow the INI-file format and, in JF, feature the extension `.ini`.

4.2. Globally used modules

JF-V1.0 requires that all trees representing composite point objects are identical and of height at most two. Furthermore, in the NVT physical ensemble, the particle number, system size and temperature remain unchanged throughout each run. After initialization, as specified in the configuration file, these parameters are stored in the JF `setting` package and the modules therein, which may be imported by all other modules, which can then autonomously construct identifiers. Helper functions for periodic boundary conditions (if available) and for the sampling of random positions are also accessible.

JF-V1.0 implements hypercubic and hypercuboid setting modules. Both settings define the inverse temperature and also the attributes of all possible particle identifiers, which are broadcast directly by the `setting` package. In contrast, the parameters of the physical system are accessed only using the modules of the specific setting (for example the `setting.hypercubic_setting` module).[6] The `setting` package and its modules are initialized by classes which inherit from the abstract `Setting` class. The `HypercuboidSetting` class defines only the hypercuboid setting, the `HypercubicSetting` class, however, sets up both the `hypercubic_setting` and the `hypercuboid_setting` modules together with the `setting` package. This allows modules that are specifically implemented for a hypercuboid setting to be used with the hypercubic setting.

Each setting can implement periodic boundaries, by inheriting from the abstract `PeriodicBoundaries` class and by implementing its methods. Since many modules of JF only rely on periodic boundaries but not on the specific setting, the `setting` package gives also access to the initialized periodic boundary conditions. Similarly, a function to create a random position is broadcast by the setting package. All the configuration files in Section 5 are for a three-dimensional cubic simulation box, that is, use the hypercubic setting with `dimension = 3`.

Additional useful modules are located in the JF `base` package. The abstract `Initializer` class located in the `initializer` module enforces the implementation of an `initialize` method. This method must be called before other public methods of the inheriting class. The `strings` module provides functions to translate strings from snake to camel case and vice versa, as well as to translate a package path into a directory path. Helper functions for vectors, such as calculating the norm or the dot product, are located in the `vectors` module.

4.3. Cell systems and cell-occupancy systems

A cell-occupancy system is an instance of a class that inherits from the abstract `CellOccupancy` class, located in the activator. Any cell-occupancy system is associated with a cell system, itself an instance of a class that inherits from the abstract `Cells` class.

In JF-V1.0, the cell system consists in a regular grid of cells that are referred to through their indices. Cells can be iterated over with the `yield_cells`

[6] Attributes in the `setting` package are copied to the modules for convenience.

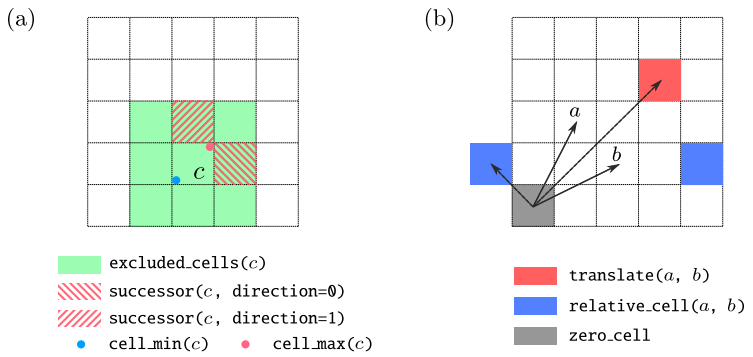


Figure 10: Cell methods. (a): `excluded_cells`, `successor`, `cell_min` and `cell_max` methods required by the abstract `Cells` class. Horizontal and vertical directions are indexed as 0 and 1, respectively. (b): `translate` and `relative_cell` methods (illustrated by vectors) required by the `PeriodicCells` class, in addition to the methods of the `Cells` class. Periodic boundary conditions are required, and the two blue cells are identical. The periodic-cell system's origin is given by the `zero_cell` property.

method. For a given cell, the excluded cells are accessed by the `excluded_cells` method, the successor cell in a suitably defined direction by the `successor` method and the lower and upper bound position in each direction through the `cell_min` and `cell_max` methods (see Fig. 10a). Finally, the `position_to_cell` method returns the cell for a given position. Cell systems with periodic boundary conditions are described as periodic cell systems (instances of classes that inherit from the abstract `PeriodicCells` class, which itself inherits from the `Cells` class). Their `zero_cell` property corresponds to the cell located at the origin. Their `relative_cell` method receives a cell and a reference cell, and establishes equivalence between the relative and the zero-cell. The inverse to this is the `translate` method (see Fig. 10b).

A cell-occupancy system (which is located in the activator) associates the identifiers of cell-based particles and of surplus particles with a cell. It also stores active cells, that is, cells that contain an active unit (see Fig. 11). Cell-based and surplus particles in the state handler correspond to units with zero velocity, so that there is no real distinction between units and particles for them. The cell-occupancy system inherits from the abstract `InternalState` class and therefore provides `__getitem__` and `update` methods. The former returns a particle identifier based on a cell, whereas the latter updates the cell occupancies based on the currently active units. This keeps the internal state consistent with the global state. Moreover the cell-occupancy may iterate over surplus particle identifiers *via* the `yield_surplus` method. The active cells and the corresponding identifiers of the active units are generated using the `yield_active_cells` method (see Fig. 11).

JF-V1.0 implements the `SingleActiveCellOccupancy` class which features only a single active cell and which keeps the active unit identifier among its

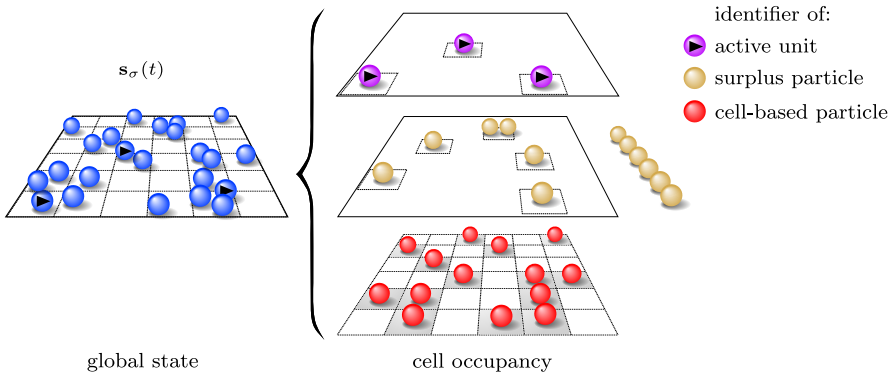


Figure 11: Cell-occupancy system, an internal state of the activator, with active units accounted for differently from surplus and cell-based particles. Only a fixed number of cell-based particle identifiers are allowed per cell (here one per cell). Surplus-particle identifiers may be iterated over from the outside of the cell-occupancy system with a `yield_surplus` method. In JF-V1.0, surplus particles form an internal dictionary mapping the cell onto the particle identifier.

private attributes. The cell-based particle identifiers are stored in an internal `_occupant` list, and surplus-particle identifiers are stored in an internal `_surplus` dictionary mapping the cell indices onto the surplus-particle identifiers.

The stored cell-occupancy system can address different levels of composite particles: one cell-occupancy system may track particles (and units) associated to root nodes, and another one particles that go with leaf nodes. This is set on initialization *via* the `cell_level` property which equals the length of the particle identifier tuple. The concerned cell system is itself set on initialization. An indicator charge allows one to select specific particles on a given level for tracking.

A single run can feature several internal states stored within the activator. These instances may rely on different cell-occupancy systems and cell systems. For consistency between internal states and the global state, each cell-occupancy system requires its own cell-boundary event handler.

4.4. Inter-particle potentials and bounding potentials

In JF, potentials play a dual role, as factor potentials U_M to event handlers but also as bounding potentials for factor potentials U_M. Potentials are located in the JF `potential` package. They inherit from the abstract `Potential` class and provide a `derivative` method. They may in addition inherit from the abstract `InvertiblePotential` class, and must then provide a `displacement` method. In JF-V1.0, derivatives and displacements are with respect to the positive change of the active unit along one of the coordinates (indicated through the `direction`). For a potential $U(\mathbf{r}_{ij})$ and `direction` $= 0$, the derivative is for example given by $[\partial/\partial x_i U(\mathbf{r}_{ij})]$.

4.4.1. Inverse-power-law potential, Lennard-Jones potential

The inverse-power-law potential (an instance of the `InversePowerPotential` class that inherits from the abstract `InvertiblePotential` class) concerns the separation vector $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ (without periodic boundary conditions, in d-dimensional space) between a unit j and an active unit i as

$$U_{(\{i,j\},\,\mathrm{inv})}(\mathbf{r}_{ij}, c_i, c_j) = c_i c_j k \left(\frac{1}{|\mathbf{r}_{ij}|}\right)^p. \tag{6}$$

Here, k and $p > 0$ correspond to the `prefactor` and `power` parameters set on initialization. The charges c_i and c_j are entered into the methods of the potential as parameters `charge_one` and `charge_two`. This allows one instance of the `InversePowerPotential` class to be used for different charges. The `derivative` method is straightforward, while the `displacement` method distinguishes the repulsive ($c_i c_j k > 0$) and the attractive ($c_i c_j k < 0$) cases.

The Lennard-Jones potential (an instance of the `LennardJonesPotential` class) implements the Lennard-Jones potential

$$U_{(\{i,j\},\,\mathrm{LJ})}(\mathbf{r}_{ij}) = k_{\mathrm{LJ}} \left[\left(\frac{\sigma}{|\mathbf{r}_{ij}|}\right)^{12} - \left(\frac{\sigma}{|\mathbf{r}_{ij}|}\right)^{6}\right], \tag{7}$$

where $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ is the separation vector (without periodic boundary conditions, in d-dimensional space) between a unit j and an active unit i. and where k_{LJ} and σ correspond to the parameters `prefactor` and `characteristic_length` set on instantiation. This Lennard-Jones potential provides a straightforward `derivative` method. Its `displacement` method relies on an algebraic inversion.

4.4.2. Displaced-even-power-law potential

An instance of the `DisplacedEvenPowerPotential` class that inherits from the abstract `InvertiblePotential` class, the displaced-even-power-law potential, concerns the separation vector $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ (without periodic boundary conditions, in d-dimensional space) between a unit j and an active unit i

$$U_{(\{i,j\},\,\mathrm{depp})}(\mathbf{r}_{ij}) = k_{\mathrm{depp}} \left(|\mathbf{r}_{ij}| - r_0\right)^p, \tag{8}$$

where $k_{\mathrm{depp}} > 0$, $p \in \{2, 4, 6, \dots\}$, and r_0, respectively, are the parameters `prefactor`, `power`, and `equilibrium_separation` parameters set on instantiation. The `derivative` and `displacement` methods are provided analytically.

4.4.3. Merged-image Coulomb potential and bounding potential

An instance of the `MergedImageCoulombPotential` class that inherits from the abstract `Potential` class, the merged-image Coulomb potential is defined for a separation vector $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$ (with periodic boundary conditions, in three-dimensional space) between a unit j and an active unit i as

$$U_{\mathrm{C}}(\mathbf{r}_{ij}, c_i, c_j) = \sum_{\mathbf{n}} c_i c_j / |\mathbf{r}_{ij} + \mathbf{n}\mathbf{L}|, \quad \mathbf{n} \in \mathbb{Z}^3, \tag{9}$$

where $\mathbf{L} = (L_x, L_y, L_x)$ are the sides of the three-dimensional simulation box with periodic boundary conditions. The conditionally convergent sum in eq. (9) can be consistently defined in terms of "tin-foil" boundary conditions [21]. It then yields an absolutely convergent sum, partly in real space and partly in Fourier space (see [9, Sect. IIIA]),

$$\psi(\mathbf{r}_{ij}, c_i, c_j) = c_i c_j \left[\sum_{\mathbf{n}\in\mathbb{Z}^3} \frac{\operatorname{erfc}(\alpha|\mathbf{r}_{ij} + \mathbf{n}L|)}{|\mathbf{r}_{ij} + \mathbf{n}L|} + \frac{4\pi}{L^3} \sum_{\mathbf{q}\neq(0,0,0)} \frac{\mathrm{e}^{-\mathbf{q}^2/(4\alpha^2)}}{\mathbf{q}^2} \cos\left(\mathbf{q}\cdot\mathbf{r}_{ij}\right) \right], \tag{10}$$

with α a tuning parameter and $\mathbf{q} = 2\pi\mathbf{m}/L$, $\mathbf{m} \in \mathbb{Z}^3$. JF-V1.0 provides this class for a cubic simulation box, with parameters that are optimized to reach machine precision for its `derivative` method. Summations over $\mathbf{n}$ and $\mathbf{m}$ are taken within spherical cutoffs, namely for all $|\mathbf{n}| \leq$ `position_cutoff` and $|\mathbf{m}| \leq$ `fourier_cutoff` except that $\mathbf{m} = (0, 0, 0)$. (The potential in eq. (10) differs from the tin-foil Coulomb potential in a constant self-energy term that does not influence the derivatives.)

The merged-image Coulomb potential is not invertible. When it serves as a factor potential, bounding potentials provide the required `displacement` method. JF-V1.0 provides a merged-image Coulomb bounding potential as an instance of the `InversePowerCoulombBoundingPotential` class, with

$$U_{\mathrm{C,Bounding}} = c_i c_j k_b / |\mathbf{r}_{ij,0}|. \tag{11}$$

Here, $\mathbf{r}_{ij,0}$ is the minimum separation vector, that is, the vector between $\mathbf{r}_i$ and the closest image of $\mathbf{r}_j$ under the periodic boundary conditions. (The merged-image Coulomb bounding potential thus involves no sum over periodic images.) The constant k_b is chosen as

$$k_b = \max_{\mathbf{r}\in[-L/2,L/2]^3} \frac{|\mathbf{r}|^3}{x} \frac{\partial\psi(\mathbf{r})}{\partial x}, \tag{12}$$

so that the factor-potential event rate is bounded. A constant $k_b \gtrsim 1.5836$ (the parameter `prefactor`) is appropriate for a cubic simulation box. The merged-image Coulomb bounding potential is closely related to the inverse-power-law potential of eq. (6) with $p = 1$, although the restriction to the minimum separation vector makes that the latter cannot be used directly.

4.4.4. Cell-based bounding potential

A cell-based bounding potential is an instance of a class that inherits from the abstract `InvertiblePotential` class. It bounds the derivative of the factor potential inside certain cell regions by constants. These constants can be computed analytically on demand or even sampled using a separate Monte Carlo algorithm. On initialization, a cell-based bounding potential receives an estimator (see Section 4.6). Also the information about the cell system is transmitted. Then, the cell-based bounding potential iterates over all pairs of cells (making use of periodic boundary conditions) and determines an upper and lower bound

derivative for the factor units being in those cells for each possible direction of motion using the estimator. Here, the cell-based bounding potential is not applied to excluded cells, where the cell-bounded event rate diverges, is simply too large, or otherwise inappropriate.

The constant-derivative bound leads to a piecewise linear invertible bounding potential. The call of the `displacement` method is accompanied by the direction of motion, the charge product, the sampled potential change and the cell separation. In JF-V1.0, any cell-based bounding potential requires a cell-boundary event handler, that detects when the displacement proposed by the `displacement` method in fact takes place outside the cell for which it is computed.

4.4.5. Three-body bending potential

The SPC/Fw water model of Section 5.3 includes a bending potential (an instance of the `BendingPotential` class), which describes the fluctuations in the bond angle within each molecule. For the three units i, j, and k within such a molecule in three-dimensional space (with j being the oxygen), it is given by

$$U_{(\{i,j,k\},\,\mathrm{bending})}(\mathbf{r}_{ij},\mathbf{r}_{jk}) = \frac{1}{2}k_b\Big(\phi_{\{i,j,k\}}(\mathbf{r}_{ij},\mathbf{r}_{jk}) - \phi_0\Big)^2. \tag{13}$$

Here, $\phi_{\{i,j,k\}}(\mathbf{r}_{ij},\mathbf{r}_{jk})$ denotes the internal angle between the two hydrogen–oxygen legs. The constants k_b and ϕ_0 are set on initialization of the potential (see [9]). The `derivative` method is provided explicitly for this potential, which is however not invertible.

In JF-V1.0, the associated bounding potential is constructed dynamically by an event handler[7] which dynamically constructs a piecewise linear bounding potential. Here, the event handler speculates on a constant bounding event rate through its position between two subsequent time-sliced positions of the active unit: $q_{\mathrm{bounding}} = \max\{q(\mathbf{r}), q(\mathbf{r}+\mathbf{v}\Delta t)\} + \mathrm{const}$ where $q(\mathbf{r})$ is the potential derivative at $\mathbf{r}$. The interval length $|\mathbf{v}\Delta t|$ and the constant offset are input from the configuration file. Fine-tuning provides an efficient bounding potential that does not under-estimate the event rate, yet limits the ratio of unconfirmed events.

4.5. Lifting schemes

Event handlers with more than two independent units require a lifting scheme (an instance of a class that inherits from the `Lifting` class). The event handler calls a method of the lifting scheme to compute its out-state. At first, the event handler prepares factor derivatives of relevant time-sliced units. The derivative table (see [9, Figs 2 and 10]) is filled with unit identifiers, factor derivatives and activity information through its `insert` method. Finally, the event handler calls

[7] instance of the `FixedSeparationsEventHandlerWithPiecewiseConstantBoundingPotential` class

the `get_active_identifier` method that returns the identifier of the next independent active unit. The lifting scheme's `reset` method deletes the derivative table. It is called before the first derivative is inserted. JF-V1.0 implements the ratio, inside-first and outside-first lifting schemes for a single independent active unit (see [9, Sect. IV]).

4.6. Estimator

Estimators (instances of a class that inherits from the abstract `Estimator` class) determine upper and lower bounds on the factor derivative in a single direction between a minimum and maximum corner of a hypercuboid for the possible separations. For this, they provide the `derivative_bound` method. Both upper and lower bounds are useful when the potential can have either positive and negative charge products (as happens for example for the merged-image Coulomb potential as a function of the two charges). In general, an estimator compares the factor derivatives for different separations in the hypercuboid to obtain the bounds. These are corrected by a prefactor and optionally by an empirical bound, which are set on instantiation (together with the factor potential).

JF-V1.0 provides estimators which either regard regularly or randomly sampled separations within the hypercuboid. The inner-point and boundary-point estimators vary the separation evenly within the hypercuboid or on the edge of the hypercuboid, respectively. For these separations, the factor potential derivatives (optionally including charges) are compared. Two more estimators consider the interaction between a charged active unit and two oppositely charged target units within a dipole. Here, the factor derivative is summed for the two possible active-target pairs. A Monte-Carlo estimator distributes both the separation and the dipole orientation randomly. The dipole-inner-point estimator varies the separations evenly but aligns the dipole orientation along the direction of the gradient of the factor derivative. The implemented estimators are appropriate for the cookbook examples of Section 5, where the upper and lower bounds on the factor derivatives (and equivalently on the event rates) must be computed for a small number of cell pairs only.

5. JF Cookbook

The configuration files[8] in JF-V1.0 introduce to the key features of the application by constructing runs for two charged point masses, for two interacting dipoles of charges, and for two interacting water molecules (using the SPC/Fw model). All configuration files are for a three-dimensional cubic simulation box with periodic boundary conditions, and they reproduce published data [9].

As specified in their `[Run]` sections, the configuration files use a single-process mediator (an instance of the `SingleProcessMediator` class), and the

[8]Configuration files in the `src/config_files/2018_JCP_149_064113` directory tree are described in this section.

(a)

```
class SingleProcessMediator:
       def __init__(self, input_output_handler:  InputOutputHandler, state_handler:  StateHandler,
                          scheduler:  Scheduler, activator:  Activator):
```

(b)

```
[section] property value
```

```
[Run]
mediator = single_process_mediator
setting = hypercubic_setting

[HypercubicSetting]
system_length = 1
beta = 2
dimension = 3

[SingleProcessMediator]
state_handler = tree_state_handler
scheduler = heap_scheduler
activator = tag_activator
input_output_handler = input_output_handler

[TagActivator]
taggers =
         coulomb (factor_type_map_in_state_tagger),
         sampling (no_in_state_tagger),
         end_of_chain (no_in_state_tagger),
         start_of_run (no_in_state_tagger),
         end_of_run (no_in_state_tagger)

[Coulomb]
event_handler = two_leaf_unit_bounding_potential
                _event_handler

[TwoLeafUnitBoundingPotentialEventHandler]
potential = merged_image_coulomb_potential
bounding_potential = inverse_power_coulomb
                     _bounding_potential

[Sampling]
event_handler = fixed_interval_sampling_event_handler

[FixedIntervalSamplingEventHandler]
sampling_interval = 0.56789
output_handler = separation_output_handler

[EndOfChain]
event_handler = same_active_periodic_direction
                _end_of_chain_event_handler

[SameActivePeriodicDirectionEndOfChainEventHandler]
chain_length = 0.78965

[TreeStateHandler]
physical_state = tree_physical_state
lifting_state = tree_lifting_state

[InputOutputHandler]
output_handlers = separation_output_handler
input_handler = random_input_handler
```

Figure 12: Configuration file `coulomb_atoms/power_bounded.ini`. (a): A typical `__init__` method of a JF class. (b): Excerpts of the configuration file (some lines split for clarity). Sections with properties and values that correspond to the argument names in the `__init__` methods of JF classes.

`setting` package is initialized by an instance of the `HypercubicSetting` class (see for example Fig. 12). All configuration files in the directory use a heap scheduler (an instance of the `HeapScheduler` class), a tree state handler (instance of the `TreeStateHandler` class), as well as an tag activator (an instance of the `TagActivator` class) in order to activate event handlers, trash candidate events and prepare in-states.

The `start_of_run`, `end_of_run`, `end_of_chain`, and `sampling` event handlers (that realize common pseudo-factors) are implemented in largely analogous sections across all the configuration files, although their parent sections (that define the corresponding taggers) provide different tag lists for trashing and activation of event handlers. The corresponding tagger sections are presented in detail in Section 5.1.1, and only briefly summarized thereafter.

5.1. Interacting atoms

The configuration files in the `coulomb_atoms` directory of JF-V1.0 implement the ECMC sampling of the Boltzmann distribution for two identical charged point masses. They interact with the merged-image Coulomb pair potential and are described by a Coulomb pair factor. One of the two point masses is active, and it moves either in $+x$, in $+y$, or in $+z$ direction. Statistically equivalent output is obtained for the merged-image Coulomb pair potential (the factor potential) associated with the inverse-power bounding potential (Section 5.1.1), or else with a cell-based bounding potential, either realized directly (Section 5.1.2), or through a cell-veto event handler (Section 5.1.3). Although the configuration files use the language of Section 1.2 for the representation of particles, all trees and branches are trivial, and each root node is also a leaf node.

5.1.1. Atomic factors, inverse-power Coulomb bounding potential

The configuration file `coulomb_atoms/power_bounded.ini` implements a single Coulomb pair factor with the merged-image Coulomb factor potential that is associated with its inverse-power Coulomb bounding potential. The same event handler realizes this factor for any separation of the point masses. The activator requires no internal state.

Although it would be feasible to directly implement (that is, hard-wire) all event handlers for this simple system, the tag activator is used. All event handlers are thus accessed *via* taggers that are listed, together with their tags, in the `[TagActivator]` section (see Fig. 13 for a tree representation of the sections). The `coulomb` tagger is an instance of the `FactorTypeMapInStateTagger` class, indicating that its event handlers require a specific in-state created from a pattern stored in a file indicated in the `[FactorTypeMaps]` section. This pattern mirrors the factor index sets and factor types for a system with two root nodes (see eq. (2)). The entry `[0, 1], Coulomb` in this file indicates that, for two point masses, a Coulomb potential would act between particles 0 and 1. From this information, the tagger's `yield_identifiers_send_event_time` method generates all the in-state identifier for any number of point masses.

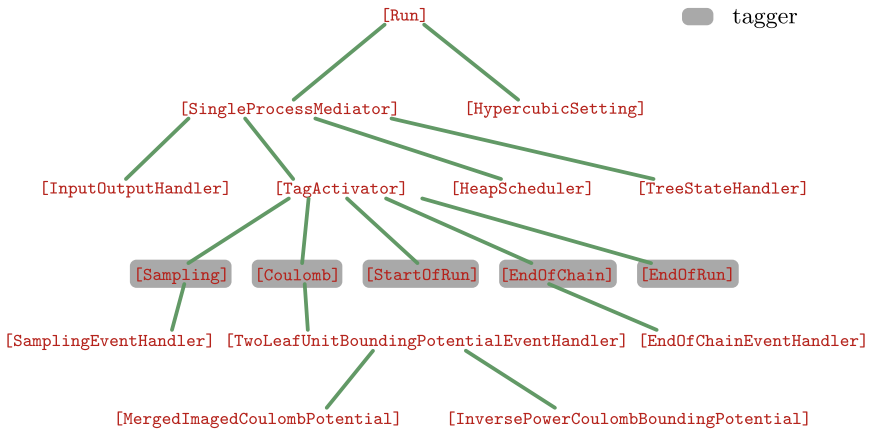


Figure 13: Tree representation the sections in the configuration file `coulomb_atoms/power_bounded.ini`. Only part of the tree is shown and names of event handlers for sampling and end-of-chain are shortened. The children of the `[TagActivator]` section correspond to all the declared taggers, which point towards sections for their associated event-handler classes.

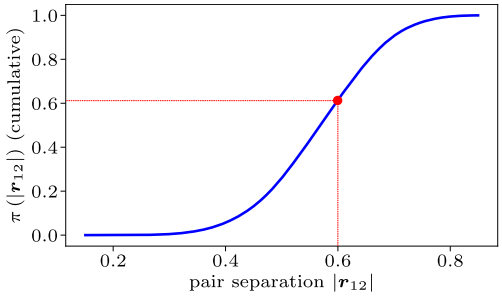
The `[Coulomb]` section specifies input for the `coulomb` tagger's tag lists (the `creates` list and the `trashes` list). Here, a `coulomb` event creates and trashes only `coulomb` candidate events (see the configuration file of Section 5.3.1 for different tag lists for the same `coulomb` event handlers).

The `[Coulomb]` section further specifies that the `coulomb` event handler is an instance of the `TwoLeafUnitBoundingPotentialEventHandler` class and that, for two point masses, only one `coulomb` event handler is needed. The corresponding section[9] specifies the factor potential to be an instance of the `MergedImageCoulombPotential` class. It specifies the bounding potential as an instance of the `InversePowerCoulombBoundingPotential` class. The `sampling`, `end_of_chain`, `start_of_run` and `end_of_run` taggers are all instances of the `NoInStateTagger` class (their event handlers require no in-state), and also provide their event handlers and their tag lists, which are then transmitted to the tag activator. Each of these taggers' `yield_identifiers_send_event_time` methods yields the in-state identifiers needed by the taggers' event handlers in order to realize corresponding factors or pseudo-factors.

The configuration file's `[InputOutputHandler]` section specifies the input-output handler. It consists of the separation-output handler (an instance of the `SeparationOutputHandler` class), which is connected to the `sampling` event handler. In the present example, it samples the nearest-image separation (under periodic boundary conditions) of any two point masses. The

[9]The `[TwoLeafUnitBoundingPotentialEventHandler]` section. The section name may be replaced by an alias to respect the tree structure of the configuration file (see Section 5.2.1).

initial global physical state is created randomly by the random-input handler (an instance of the `RandomInputHandler` class). The configuration file `coulomb_atoms/power_bounded.ini` reproduces published data (see Fig. 14,②).



method	$\pi(\lvert\mathbf{r}_{12}\rvert < 0.6)$
①	0.61305 ± 0.00003
②	0.61306 ± 0.00006
③	0.61299 ± 0.00008
④	0.61307 ± 0.00012

Figure 14: Cumulative histogram of the pair separation $|\mathbf{r}_{12}|$ (nearest image) for two charges in a periodic three-dimensional cubic simulation box with periodic boundary conditions ($\beta c_1 c_2 = 2$, $L = 1$). ①: Reversible Markov-chain Monte Carlo (see [9, Fig. 8]) ②: Method of Section 5.1.1 ③: Method of Section 5.1.2 ④: Method of Section 5.1.3 , each with standard errors for $\pi(|\mathbf{r}_{12}| < 0.6)$.

The configuration file `coulomb_atoms/power_bounded.ini` can be modified for N point masses. In the `[RandomInputHandler]` section, the number of root nodes must then equal N. In the `[Coulomb]` section, the number of event handlers must be set to at least $N-1$ (this instructs the `Coulomb` tagger to deep-copy the required number of event handlers). Without changing the factor-type map with respect to the $N = 2$ case, each event handler which will be presented with the correct in-state corresponding to a pair of units with one of them being the active unit. The complexity of the implemented algorithm is $\mathcal{O}(N)$ per event.

5.1.2. Atomic factors, cell-based bounding potential

The configuration file `coulomb_atoms/cell_bounded.ini` implements a single Coulomb pair factor with the merged-image Coulomb potential, just as the configuration file of Section 5.1.1. However, a cell-occupancy internal state associates the factor potential with a cell-based bounding potential. The target (non-active) unit may be cell-based or surplus (see Fig. 11). The target unit may also be in an excluded nearby cell of the active cell (see Fig. 10), for which the cell-based bounding potential cannot be used. In consequence, three taggers correspond to distinct event handlers that together realize the Coulomb pair factor. The consistency requirement of JF-V1.0 assures that particles and units are always associated with the same cell.

Taggers and their tags are listed in the `[TagActivator]` section. The `coulomb_cell_bounding` tagger, for example, appears as an instance of the `CellBoundingPotentialTagger` class. The `coulomb_cell_bounding` event handler then realizes the Coulomb factor unless the cell of the target particle is excluded with respect to the active cell and unless it is a surplus particle (in

these cases the tagger does not generate any in-state for its event handler). Otherwise, the Coulomb pair factor is realized by a `coulomb_surplus`-tagged event handler or by a `coulomb_nearby` event handler. (For two units, as the active unit is taken out of the cell-occupancy system, no surplus candidate events are ever created.)

The cell-occupancy systems (an instance of the `SingleActiveCellOccupancy` class) is also declared in the `[TagActivator]` section and further specified in the `[SingleActiveCellOccupancy]` section. The associated cell system is described in the `[CuboidPeriodicCells]` section. The internal state, set in the `[SingleActiveCellOccupancy]` section, has no charge value. This indicates that the identifiers of all particles at the cell level (here `cell_level` = 1) are tracked (see Section 5.3.2 for an example where this is handled differently).

The `coulomb_nearby` tagger, an instance of the `ExcludedCellsTagger` class, yields the identifiers of particles in excluded cells of the active cell, by iterating over cells and by checking whether they contain appropriate identifiers. The `coulomb_surplus` tagger similarly relies on the `yield_surplus` method of the cell-occupancy system to generate in-states.

To keep the internal state consistent with the global state, a cell-boundary event handler is used in the `CellBoundaryTagger` class (together, this builds `cell_boundary` candidate events). The cell-boundary tagger just yields the active-unit identifier as the in-state used in the corresponding event handler. The configuration file `coulomb_atoms/cell_bounded.ini` reproduces published data (see Fig. 14,③).

To adapt the configuration file for $N > 2$ point masses (from the $N = 2$ case that is provided), in the `[RandomInputHandler]`, `number_of_root_nodes` must be set to N. The number of `coulomb_cell_bounding`, `coulomb_nearby`, and `coulomb_surplus` event handlers must be increased. Surplus particles can now exist. The number of event handlers to allow for depends on the cell system, whose parameters must be adapted in order to limit the number of surplus particles, and also to retain useful cell-based bounds for the Coulomb event rates.

5.1.3. Atomic factors, cell-veto

The configuration file `coulomb_atoms/cell_veto.ini` implements a Coulomb pair factor together with the merged-image Coulomb potential. A cell-occupancy internal state is used. The Coulomb pair factor is then realized, among others, by a cell-veto event handler, which associates the merged-image Coulomb potential with a cell-based bounding potential.

All the Coulomb pair factors of the active particle with target particles that are neither excluded nor surplus are taken together in a set of Coulomb factors, and realized by a single `coulomb_cell_veto` event handler. The candidate event-time can be calculated with the branch of the active unit as the in-state, which is implemented in the `CellVetoTagger` class. (The cell-veto tagger returns the identifier of the active unit.) The event handler returns the target cell (in which the candidate unit is to be localized) together with the candidate event time. The out-state request is accompanied by the branch of the target unit (if it

exists), and the out-state computation is in analogy with the case studied in Section 5.1.2.

The configuration file features the `coulomb_cell_veto` tag together with the `coulomb_nearby`, `coulomb_surplus`, `cell_boundary`, `sampling`, `end_of_chain`, `start_of_run`, and `end_of_run` tags. The configuration file reproduces published data (see Fig. 14,④).

To adapt the configuration file for N point masses, the number of root nodes must be set to N in the `[RandomInputHandler]` section. The number of event handlers for the `coulomb_nearby` and `coulomb_surplus` events might have to be increased. However, a single cell-veto event handler realizes any number of factors with cell-based target particles whereas in Section 5.1.2 each of them required its own event handler.

5.2. Interacting dipoles

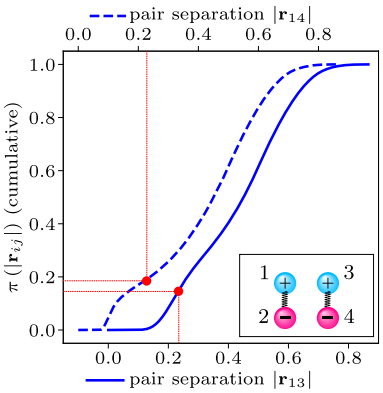
The configuration files in the `dipoles` directory of JF-V1.0 implement the ECMC sampling of the Boltzmann distribution for two identical finite-size dipoles, for a model that was introduced previously [9]. Point masses in different dipoles interact *via* the merged-image Coulomb potential (pairs $1-3$, $1-4$, $2-3$, $2-4$ in Fig. 15). Point masses within each dipole interact with a short-ranged potential (pairs $1-2$ and $3-4$). A repulsive short-range potential between oppositely charged atoms in different dipoles counterbalances the attractive Coulomb potential at small distances (pairs $1-4$ and $2-3$).

Each dipole is a composite point object made up of two oppositely charged point masses. It is represented as a tree with one root node that has two children. The number of root nodes in the system is set in the `[RandomInputHandler]` section of the configuration file, where the dipoles are created randomly through the `fill_root_node` method in the `DipoleRandomNodeCreator` class. In the `setting` package, the input handler specifies that there are two root nodes (`number_of_root_nodes` $= 2$). Each of them contains two nodes (which is coded as `number_of_nodes_per_root_node` $= 2$) and the number of node levels is two (`number_of_node_levels` $= 2$). As this numbers are set in the `setting` package, all the JF modules can autonomously construct all possible particle identifiers.

Statistically equivalent output is obtained for pair factors for all interactions (Section 5.2.1), for dipole–dipole Coulomb factors and their factor potential associated with a cell-based bounding potential (Section 5.2.2), for dipole–dipole Coulomb factors with the cell-veto algorithm (Section 5.2.3), and by alternating between concurrent moves of the entire dipoles with moves of the individual point masses (Section 5.2.4). The latter example showcases the collective-motion possibilities of ECMC integrated into JF. All configuration files here implement the short-ranged potential as an instance of the `DisplacedEvenPowerPotential` class with `power` $= 2$ and the repulsive short-range potential as an instance of the `InversePowerPotential` class with `power` $= 6$.

5.2.1. Atomic Coulomb factors

The configuration file `dipoles/atom_factors.ini` implements for each concerned pair of point masses a Coulomb pair factor, with the merged-image



method	$\pi(\lvert\mathbf{r}_{13}\rvert < 0.22)$	$\pi(\lvert\mathbf{r}_{14}\rvert < 0.22)$
①	0.12128 ± 0.00003	0.18453 ± 0.00003
②	0.12129 ± 0.00003	0.18458 ± 0.00004
③	0.12130 ± 0.00002	0.18457 ± 0.00003
④	0.12129 ± 0.00003	0.18456 ± 0.00005
⑤	0.12108 ± 0.00037	0.18412 ± 0.00053

Figure 15: Cumulative histogram of the pair separation $|\mathbf{r}_{13}|$ and $|\mathbf{r}_{14}|$ (nearest image) for two dipoles (see the inset) in a periodic three-dimensional cubic simulation box with periodic boundary conditions ($\beta c_i c_j = \pm 1$, $L = 1$). ①: Reversible Markov-chain Monte Carlo (see [9, Fig. 11]) ②: Method of Section 5.2.1 ③: Method of Section 5.2.2 ④: Method of Section 5.2.3 ⑤: Method of Section 5.2.4, each with standard errors for $\pi(|\mathbf{r}_{13}| < 0.22)$ and $\pi(|\mathbf{r}_{14}| < 0.22)$.

Coulomb potential associated with the the inverse-power Coulomb bounding potential. Several event handlers that are instances of the same class realize these factors, and the number of event handlers must scale with their number. No internal state is declared. Pair factors are implemented for each pair of point masses that interact with a harmonic or a repulsive potential. One of the four point masses is active at each time, and it moves either in $+x$, in $+y$, or in $+z$ direction. The configuration file represents composite point objects as trees with two levels (see Section 1.2). Positions and velocities are kept consistent on both levels, although the root-unit properties are not made use of. The tree structure only serves to identify leaf units on the same dipole.

In the configuration file, taggers and tags are listed in the `[TagActivator]` section. The `coulomb`, `harmonic`, and `repulsive` taggers are separate instances of the same `FactorTypeMapInStateTagger` class, and the corresponding sections set up the corresponding event handlers. Both the `harmonic` and the `repulsive` event handlers are instances of the `TwoLeafUnitEventHandler` class. Aliasing nevertheless assures a tree-structured configuration file (the `harmonic` tagger is for example declared with a `HarmonicEventHandler` class which is an alias for the `TwoLeafUnitEventHandler` class). The `coulomb` tagger and its event handlers are treated as in Section 5.1.1.

The sampling, start-of-run, end-of-run and end-of-chain pseudo-factors are realized by event handlers that are set up in the same way as in all other configuration files. However, the parent sections differ: the parent of the `[InitialChainStartOfRunEventHandler]` section sets the `start_of_run` tagger, which specifies that after the `start_of_run` event, new `coulomb`, `harmonic`, `repulsive`, `sampling` `end_of_chain`, and `end_of_run` event handlers must be activated. The tag lists thus differ from those of the `[StartOfRun]` section

in other configuration files. The configuration file `dipoles/atom_factors.ini` reproduces published data (see Fig. 15,②).

5.2.2. Molecular Coulomb factors, cell-based bounding potential

The configuration file `dipoles/cell_bounded.ini` implements for each pair of dipoles a Coulomb four-body factor. (The sum of the merged-image Coulomb potentials for pairs $1-3$, $1-4$, $2-3$, $2-4$ in Fig. 15 constitutes the Coulomb factor potential.) The event rates for such factors decay much faster with distance than for Coulomb pair factors, and the chosen lifting scheme considerably influences the dynamics (see [9, Sect. IV]). The configuration file installs a cell-occupancy internal state on the dipole level (rather than for the point masses). A cell-bounded event handler then realizes a Coulomb four-body factor with its factor potential associated with an orientation-independent cell-based bounding potential for dipole pairs that are not in excluded cells relative to each other. The configuration file furthermore implements pair factors for the harmonic and the repulsive interactions. One of the four point masses is active at each time, and it moves either in $+x$, in $+y$, or in $+z$ direction.

The configuration file's `[TagActivator]` section defines all taggers and their corresponding tags. Among the taggers for event handlers realizing the Coulomb four-body factor, the `coulomb_cell_bounding` tagger differs markedly from the set-up in Section 5.1.2, as the event handler[10] is for a pair of composite point objects. The lifting scheme is set to `inside_first_lifting`. The bounding potential is defined in the `[CellBoundingPotential]` section. A dipole Monte Carlo estimator is used for simplicity (see Section 4.6). As it obtains an upper bound for the event rate from random trials for each relative cell orientations, its use is restricted to there being only a small number of cells. The `coulomb_nearby` and `coulomb_surplus` taggers are for event handlers realizing the Coulomb four-body factor when the bounding potential cannot be used. In this case, the merged-image Coulomb potential is summed for the factor potential, but also for the bounding potential.[11] The standard `sampling`, `end_of_chain`, `end_of_run`, `start_of_run` taggers as well as the ones responsible for the harmonic and repulsive potentials are set up in a similar way as in Section 5.2.1.

The `[TagActivator]` section defines the internal state that is used by the `coulomb_cell_bounding`, `coulomb_nearby`, and `coulomb_surplus` taggers. The `[SingleActiveCellOccupancy]` section specifies the cell level (`cell_level` $= 1$ indicates that the particle identifiers have length one, corresponding to root nodes, rather than length two, which would correspond to the dipoles' leaf nodes). Positions and velocities must thus be kept consistent on both levels. The cell-occupancy system requires the presence of a `cell_boundary` event handler, again on the level of the root nodes. This event handler is aware of

[10] set in the `[TwoCompositeObjectCellBoundingPotentialEventHandler]` section

[11] The tree structure of the configuration file is hidden in this case, as the JF factory (which builds instances of classes based on its content) creates separate instances for all the descendants of a section, not requiring the use of aliases.

the cell level, and it ensures consistency of the events triggered by the cell-based bounding potential with the underlying cell system. The configuration file `dipoles/cell_bounded.ini` reproduces published data (see Fig. 15,③).

5.2.3. Molecular Coulomb factors, cell-veto

The configuration file `dipoles/cell_veto.ini` implements the same factors and pseudo-factors and the same internal state as the configuration file of Section 5.2.2. A single cell-veto event handler then realizes the set of factors that relate to cells that are not excluded for any number of cell-based particles, whereas in the earlier implementation, the number of cell-bounded event handlers must exceed the possible number of particles in non-excluded cells of the active cell. This is what allows to implement ECMC with a complexity of $\mathcal{O}(1)$ per event.

The configuration file resembles that of Section 5.2.2. It mainly replaces the latter file's `coulomb_cell_bounding` event handlers with a `coulomb_cell_veto` event handler. Slight differences reflect the fact that a cell-veto event handler uses no `displacement` method of the bounding potential but obtains the displacement from the total event rate (see the discussion in Section 3.1.3). The configuration file `dipoles/cell_veto.ini` reproduces published data (see Fig. 15,④).

5.2.4. Atomic Coulomb factors, alternating root mode and leaf mode

The configuration file `dipoles/dipole_motion.ini` implements two different modes. In leaf mode, at each time one of the four point masses is active, and it moves either in $+x$, in $+y$, or in $+z$ direction (see Fig. 16a). In root mode, at each time the point masses of one dipole moves as a rigid block, in the same direction (see Fig. 16b). (The root mode, by itself, does not assure irreducibility of the Markov-chain algorithm, as the orientation and shape of any dipole molecule would remain unchanged throughout the run.)

JF-V1.0 represents the dipoles as trees, and both modes are easily implemented. In leaf mode, the Coulomb factors are realized by `coulomb_leaf` event handlers that are instances of the same class[12] as the `coulomb_nearby` event handlers in Sections 5.2.2 and 5.2.3. The root mode, in turn, is patterned after the simulation of two point masses (as in Section 5.1.1): all inner-dipole potentials are constant. The inter-dipole Coulomb potentials sum up to an effective two-body potential, the factor potential of a two-body factor realized in a Coulomb-dipole event handler. The repulsive short-range potential between oppositely charged atoms in different dipoles also translates into a potential between the dipoles in rigid motion, and serves as a factor potential of a two-body factor, realized in a specific event handler.

Taggers and their tags are listed in the `[TagActivator]` section. The `harmonic_leaf`, `repulsive_leaf` (leaf-mode) taggers, as well as all those related to event handlers that realize pseudo-factors are as in Section 5.2.1. The

[12]Instances of the `TwoCompositeObjectSummedBoundingPotentialEventHandler` class

`coulomb_leaf` tagger corresponds to the `coulomb_nearby` tagger in Section 5.2.2. The `coulomb_root` and `repulsive_root` taggers are analogous to those in Section 5.1.1 for the two-atom case.

As all other operations that take place in JF, the switches between leaf mode and root mode are also formulated as events. They are related to two pseudo-factors and realized by a `leaf_to_root` event handler and by a `root_to_leaf` event handler, respectively. (These two event handlers are aliases for instances of the `RootLeafUnitActiveSwitcher` class.) The `root_to_leaf` and `leaf_to_root` taggers, in addition to the `create` and `trash` lists, set up separate `activate` and `deactivate` lists (see Section 2.4). The configuration file reproduces published data (see Fig. 15,⑤). Of particular interest is that the tree representation of composite point objects keeps consistency between leaf-node units and root-node units: the event handlers return branches of cnodes for all independent units (see Fig. 7) whose unit information can be integrated into the global state.

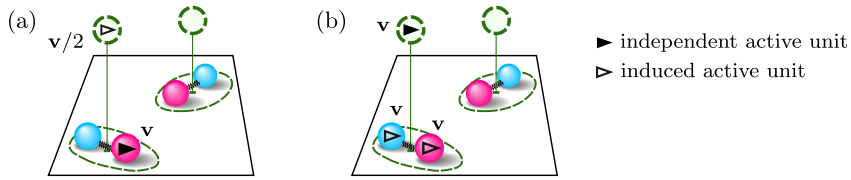


Figure 16: Two moves implemented in `dipoles/dipole_motion.ini`. (a): In leaf mode, a single independent active leaf unit has velocity $\mathbf{v}$. The corresponding dipole center (the active root unit) is induced to move at $\mathbf{v}/2$. (b): In root mode, one dipole (independent active root unit) has velocity $\mathbf{v}$, and both its active leaf units have induced velocity $\mathbf{v}$.

5.3. Interacting water molecules (SPC/Fw model)

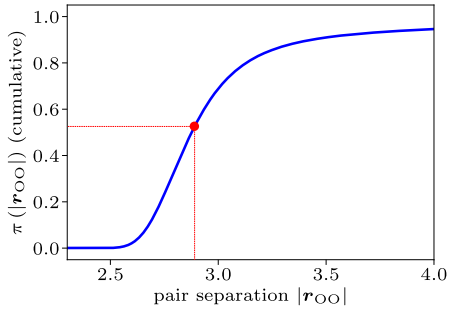
The configuration files in the `water` directory implement the ECMC sampling of the Boltzmann distribution for two water molecules, using the SPC/Fw model that was previously studied with ECMC [9]. Molecules are represented as composite point objects with three charged point masses, one of which is positively charged (representing the oxygen) and the two others are negatively charged (representing the hydrogens). Point masses in different water molecules interact *via* the merged-image Coulomb potential. In addition, point masses within each molecule interact with a three-body bending interaction, and a harmonic oxygen–hydrogen potential. Finally, any two oxygens interact through a Lennard-Jones potential [9].

In the tree state handler (defined in the `[TreeStateHandler]` section, a child of the `[SingleProcessMediator]` section), water molecules are represented as trees with a root node and three children (the leaf nodes of the tree). The total number of water molecules (that is, of root nodes) is set in the `[RandomInputHandler]` section of each configuration file. The molecules are created through the `fill_root_node` method in the `WaterRandomNodeCreator` class. There are two node levels (`number_of_node_levels` = 2) and three

nodes per root node (`number_of_nodes_per_root_node` = 3). The charges of a molecule are set in the `[ElectricChargeValues]` section (a descendant of the `[WaterRandomNodeCreator]` section).

All the configuration files in the `water` directory of JF-V1.0 implement the pair harmonic factors that are realized through `harmonic` event handlers. The corresponding taggers are defined in the `[Harmonic]` sections, with the displaced even-power potential and its parameters set in the `[HarmonicEventHandler]` and `[HarmonicPotential]` sections. The configuration files furthermore implement the taggers corresponding to the three-body bending factors in their `[Bending]` sections. The `bending` event handler has three independent units (attached to branches). It thus requires a lifting scheme (which is chosen in the `[BendingEventHandler]` section), which is however unique (see [9, Fig. 2]). In all these configuration files, one of the six point masses is active, and it moves either in $+x$, in $+y$, or in $+z$ direction (the optional rigid displacement of the entire water molecule, could be set up as in Section 5.2.4).

Statistically equivalent output is obtained for a simple set-up featuring pair factors for the Coulomb potential and a Lennard-Jones interaction that is inverted (Section 5.3.1), or for a molecular-factor Coulomb potential associated with a power-law bounding potential and a cell-based Lennard-Jones bounding potential (Section 5.3.2). In addition, the cell-veto algorithm for the Coulomb potential coupled to an inverted Lennard-Jones potential (Section 5.3.3) is also provided. Finally, cell-veto event handlers take part in the realization of complex molecular Coulomb factors and also realize Lennard-Jones factors between oxygens (Section 5.3.4). This illustrates how multiple independent cell-occupancy systems may coexist within the same run.



method	$\pi(\lvert\mathbf{r}_{OO}\rvert < 2.9)$
①	0.53824 ± 0.00003
②	0.5385 ± 0.0003
③	0.5390 ± 0.0002
④	0.5400 ± 0.0010
⑤	0.5398 ± 0.0010

Figure 17: Cumulative histogram of the oxygen–oxygen pair separation $|\mathbf{r}_{OO}|$ for two SPC/Fw water molecules in a periodic cubic simulation box. ①: Reversible Markov-chain Monte Carlo (see [9, Fig. 14]) ②: Method of Section 5.3.1 ③: Method of Section 5.3.2 ④: Method of Section 5.3.3 ⑤: Method of Section 5.3.4, each with standard errors for $\pi(|\mathbf{r}_{OO}| < 2.9)$.

5.3.1. Atomic Coulomb factors, Lennard-Jones inverted

The configuration file `water/coulomb_power_bounded_lj_inverted.ini` implements pair Lennard-Jones, harmonic and Coulomb factors. The Coulomb

factors are realized for any distance of the point masses by event handlers that associate the merged-image Coulomb potential with its inverse-power Coulomb bounding potential. The Lennard-Jones potential is inverted. This configuration file needs no internal state.

In the configuration file, the `[TagActivator]` section lists all the taggers together with their tags, which in addition to the taggers related to pseudo-factors, are reduced to `coulomb`, `harmonic`, `bending`, and `lennard_jones`. The merged-image Coulomb potential with its associated power-law bounding potential (both for attractive and repulsive charge products) is specified in the `[Coulomb]` section of the configuration file. The Lennard-Jones potential is invertible and its `displacement` method is used rather than that of a bounding potential. The output handler is defined in the `[OxygenOxygenSeparationOutputHandler]` section, a child of the `[InputOutputHandler]` section. It obtains all the units, extracts the oxygens through their unit identifier, and records the oxygen–oxygen separation distance. This reproduces published data (see Fig. 17,②).

5.3.2. Molecular Coulomb factors, Lennard-Jones cell-bounded

The configuration file `water/coulomb_power_bounded_lj_cell_bounded.ini` for the water system corresponds to pair factors for the Lennard-Jones and the harmonic potentials and to molecular factors for the Coulomb interaction. The Coulomb factor potential is the sum of the merged-image Coulomb potential for the nine relevant pairs of point masses (pairs across two molecules). It is realized in a particular event handler,[13] analogously to how this is done for the Coulomb interaction in Sections 5.2.2 and 5.2.3. The associated bounding potential (both for attractive and repulsive charge combinations) is given by the sum over all the individual pairs. Although the Lennard-Jones interaction can be inverted, the configuration file sets up a cell-occupancy internal state that tracks the identifiers for the oxygens. As in previous cases, this leads to three types of events, corresponding to the nearby, surplus, and cell-based particles, in addition to cell-boundary events.

Taggers and their tags are listed in the `[TagActivator]` section. Taggers are generally utilized as in other configuration files. The internal state is specified in the `[TagActivator]` section. As set up in the `[SingleActiveCellOccupancy]` section, it features a `oxygen_indicator` charge (set in the `[OxygenIndicator]` section). The oxygen-indicator charge is non-zero only for the oxygens. In consequence, the oxygen cell system (defined in the `[OxygenCell]` section) tracks only oxygens. This reproduces published data (see Fig. 17,③). Nevertheless, this configuration file does not scale up easily with system size.

5.3.3. Molecular Coulomb cell-veto, Lennard-Jones inverted

The configuration file `water/coulomb_cell_veto_lj_inverted.ini` for the water system corresponds to the same factors as in Section 5.3.2. As a preliminary step towards the treatment of all long-range interactions with the cell-veto

[13] an instance of the `TwoCompositeObjectSummedBoundingPotentialEventHandler` class.

algorithm, in Section 5.3.4, molecular Coulomb factors are realized here (for non-excluded cells of the active cell) with a cell-veto event handler.

Taggers and their tags are listed in the `[TagActivator]` section, and they are generally similar to those of other configuration files. In addition, the internal state for the Coulomb system is defined in the `[TagActivator]` section and further described in the `[SingleActiveCellOccupancy]` section. The latter describes the cell level (which serves for the water molecules) as on the root node level (`cell_level` = 1), the barycenter of the leaf-node positions of each water molecule. (Root-node and leaf-node positions are set in the random input handler, which itself uses a water random node creator.)

The event handlers consistently update all leaf-node positions and root-node positions from a valid initial configuration obtained in an instance of the `WaterRandomNodeCreator` class. Consistency will be deteriorated over long runs, but this is of little importance for the simple example case presented here. The configuration file reproduces published data (see Fig. 17,④).

5.3.4. Molecular Coulomb cell-veto, Lennard-Jones cell-veto

The configuration file `water/coulomb_cell_veto_lj_cell_veto.ini` offers no new factors compared to Sections 5.3.2 and 5.3.3, but it uses, for illustration purposes, two cell-occupancy systems and two cell-veto event handlers. As nearby and surplus particles are excluded from the cell-veto treatment, this implies two sets of cell-based, nearby, and surplus event handlers in addition to two cell-boundary event handlers. For the molecular Coulomb factors, the cell-veto event handler receives as a factor potential the sum of pairwise merged-image Coulomb potentials with attractive and repulsive charge combinations. Cells track the barycenter of individual water molecules, and consistency between root-node units and leaf-node units is of importance. Although the Lennard-Jones potential can be inverted, the configuration file sets up a second cell-occupancy system for the Lennard-Jones potential. The cell-occupancy system tracks only leaf-node particles that correspond to oxygen atoms.

Taggers and their tags are listed in the `[TagActivator]` section. This section is of interest as it sets up the internal state as two cell-occupancy systems, both instances of the same `SingleActiveCellOccupancy` class. They require different parameters, and are therefore presented under aliases, in the `[OxygenCell]` and `[MoleculeCell]` sections. Each of theses cell-occupancy systems use a separate cell system instance of the same class. As the two cell systems have the same parameters, they do not need to be aliased in the configuration file. The configuration file reproduces published data (see Fig. 17,⑤).

6. Licence, GitHub repository, Python version

JF, the Python application described in this paper, is an open-source software project that grants users the rights to study and execute, modify and distribute the code. Modifications can be fed back into the project.

6.1. Licence information, used software

JF is made available under the GNU GPLv3 licence (for details see the JF `LICENCE` file). The use of the Python `MDAnalysis` package [27, 11] for reading and writing `.pdb` files, of the Python `Dill` package [25, 26] for dumping and restarting a run of the application, and of the Python `Matplotlib` [15] and `NumPy` [31, 38] packages for the graphical analysis of output is acknowledged.

6.2. GitHub repository

`JeLLyFysh`, the public repository for all the code and the documentation of the application, is part of a public GitHub organization.[14] The repository can be forked (that is, copied to an outside user's own public repository) and from there studied, modified and run in the user's local environment. Users may contribute to the JF application *via* pull requests (see the JF `README` and `CONTRIBUTING.md` files for instructions and guidelines). All communication (bug reports, suggestions) takes place through GitHub "Issues", that can be opened in the repository by any user or contributor, and that are classified in GitHub projects on `JeLLyFysh`.

6.3. Python version, coding conventions

JF-V1.0 is compatible with Python 3.5 (and higher) and with PyPy 7 (and higher), a just-in-time compiling Python alternative to interpreted CPython (see the JF documentation for details). JF code adheres to the PEP8 style guide for Python code, except for the linewidth that is set to 120 (see the `CONTRIBUTING.md` file for details).

Following the PEP8 Python naming convention, JF modules and packages are spelled in snake case and classes in camel case (the `state_handler` module thus contains the `StateHandler` class). In configuration files, section titles are in camel case and enclosed in square brackets (see Fig. 13).

Versioning of the JF project adopts two-to-four-field version numbers defined as Milestone.Feature.AddOn.Patch. Version 1.0, as described, represents the first development milestone which reproduces published data [9]. Patches and bugfixes of this version will be given number 1.0.0.1, 1.0.0.2, etc. (Finer-grained distinction between versions is obtained through the hashes of master-branch GitHub commits.) New configuration files and required extensions are expected to lead to versions 1.0.1. 1.0.2, etc. Version 1.1 is expected to fully implement different dimensions and arbitrary rectangular and cuboid shapes of the JF `potential` package. Versions 1.2 and 1.3 will consistently implement $n_{\mathrm{ac}} > 1$ independent active particles (on a single processor) and eliminate unnecessary time-slicing for some events triggered by pseudo-factors and for unconfirmed events. All development from Versions 1.0 to 2.0 can be undertaken concurrently. Fully parallel code is planned for Version 3.0. In JF development, two-field versions (2.0, 3.0, etc) may introduce incompatible code, while three- and four-field version numbers are intended to be backward compatible.

[14] The organization's url is `https://github.com/jellyfysh`

7. Conclusions, outlook

As presented in this paper, JF is a computer application for ECMC simulations that is hoped to become useful for researchers in different fields of computational science. The JF-V1.0 constitutes its first development milestone: built on the mediator design pattern, it systematically formulates the entire ECMC time evolution in terms of events, from the start-of-run to the end-of-run, including sampling, restarts (that is, end-of-chain), and the factor events. A number of configuration files validate JF-V1.0 against published test cases for long-range interacting systems [9].

For JF-V1.0, consistency has been the main concern, and code has not yet been optimized. Also, the handling of exceptions remains rudimentary, although this is not a problem for the cookbook examples of Section 5.

All the methods are written in Python. Considerable speed-up can certainly be obtained by rewriting time-consuming parts of the application in compiled languages, in particular of the `potential` package. One of the principal limitations of JF-V1.0 is that pseudo-factor-related and unconfirmed events are time-sliced, leading to superfluous trashing and re-activation of candidate events. Optimized bounding potentials for many-particle factor potentials appear also as a priority.

The consistent implementation of an arbitrary number n_{ac} of simultaneously active particles is straightforward, although it has also not been implemented fully in JF-V1.0. (As mentioned, this is planned for JF (Version 2.0)). This will enable full parallel implementations on multiprocessor machines. Simplified parallel implementations for one-dimensional systems and for hard-disk models in two dimensions are currently being prototyped. The parallel computation of candidate events (using the `MultiProcessMediator` class implemented in JF-V1.0) is at present rather slow. Bringing the full power of parallelization and of multi-process ECMC to real-world applications appears as its outstanding challenge for JF.

Acknowledgements

P.H. acknowledges support from the Bonn-Cologne Graduate School of Physics and Astronomy honors branch and from Institut Philippe Meyer. L.Q. and M.F.F. acknowledge hospitality at the Max-Planck-Institut für Physik komplexer Systeme, Dresden, Germany. M.F.F. acknowledges financial support from EPSRC fellowship EP/P033830/1 and hospitality at Ecole normale supérieure. W.K. acknowledges support from the Alexander von Humboldt Foundation.

References

[1] Alder, B.J., Wainwright, T.E., 1957. Phase Transition for a Hard Sphere System. J. Chem. Phys. 27, 1208–1209. doi:10.1063/1.1743957.

[2] Alder, B.J., Wainwright, T.E., 1959. Studies in Molecular Dynamics. I. General Method. J. Chem. Phys. 31, 459–466. doi:10.1063/1.1730376.

[3] Bannerman, M.N., Lue, L., 2010. Exact on-event expressions for discrete potential systems. J. Chem. Phys. 133, 124506–124506. doi:10.1063/1.3486567.

[4] Bernard, E.P., Krauth, W., 2011. Two-Step Melting in Two Dimensions: First-Order Liquid-Hexatic Transition. Phys. Rev. Lett. 107, 155704. URL: http://link.aps.org/doi/10.1103/PhysRevLett.107.155704, doi:10.1103/PhysRevLett.107.155704.

[5] Bernard, E.P., Krauth, W., Wilson, D.B., 2009. Event-chain Monte Carlo algorithms for hard-sphere systems. Phys. Rev. E 80, 056704. URL: http://link.aps.org/doi/10.1103/PhysRevE.80.056704, doi:10.1103/PhysRevE.80.056704.

[6] Bierkens, J., Bouchard-Côté, A., Doucet, A., Duncan, A.B., Fearnhead, P., Roberts, G., Vollmer, S.J., 2017. Piecewise Deterministic Markov Processes for Scalable Monte Carlo on Restricted Domains arXiv:1701.04244.

[7] Diaconis, P., Holmes, S., Neal, R.M., 2000. Analysis of a nonreversible Markov chain sampler. Annals of Applied Probability 10, 726–752.

[8] Ding, F., Tsao, D., Nie, H., Dokholyan, N.V., 2008. Ab Initio Folding of Proteins with All-Atom Discrete Molecular Dynamics. Structure 16, 1010–1018. URL: https://doi.org/10.1016/j.str.2008.03.013, doi:10.1016/j.str.2008.03.013.

[9] Faulkner, M.F., Qin, L., Maggs, A.C., Krauth, W., 2018. All-atom computations with irreversible Markov chains. The Journal of Chemical Physics 149, 064113. URL: https://doi.org/10.1063/1.5036638, doi:10.1063/1.5036638.

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. 1 ed., Addison-Wesley Professional.

[11] Gowers, R., Linke, M., Barnoud, J., Reddy, T., Melo, M., Seyler, S., Domański, J., Dotson, D., Buchoux, S., Kenney, I., Beckstein, O., 2016. MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations, in: Proceedings of the 15th Python in Science Conference, SciPy. URL: https://doi.org/10.25080/majora-629e541a-00e, doi:10.25080/majora-629e541a-00e.

[12] Harland, J., Michel, M., Kampmann, T.A., Kierfeld, J., 2017. Event-chain Monte Carlo algorithms for three- and many-particle interactions. EPL (Europhysics Letters) 117, 30001. URL: http://stacks.iop.org/0295-5075/117/i=3/a=30001.

[13] Hasenbusch, M., Schaefer, S., 2018. Testing the event-chain algorithm in asymptotically free models. Phys. Rev. D 98, 054502. URL: https://link.aps.org/doi/10.1103/PhysRevD.98.054502, doi:10.1103/PhysRevD.98.054502.

[14] Herbordt, M.C., Khan, M.A., Dean, T., 2009. Parallel Discrete Event Simulation of Molecular Dynamics Through Event-Based Decomposition, in: 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors, IEEE. URL: https://doi.org/10.1109/asap.2009.39, doi:10.1109/asap.2009.39.

[15] Hunter, J.D., 2007. Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering 9, 90–95. URL: https://doi.org/10.1109/mcse.2007.55, doi:10.1109/mcse.2007.55.

[16] Kapfer, S.C., Krauth, W., 2013. Sampling from a polytope and hard-disk Monte Carlo. Journal of Physics: Conference Series 454, 012031. URL: http://stacks.iop.org/1742-6596/454/i=1/a=012031, doi:10.1088/1742-6596/454/1/012031.

[17] Kapfer, S.C., Krauth, W., 2015. Two-Dimensional Melting: From Liquid-Hexatic Coexistence to Continuous Transitions. Phys. Rev. Lett. 114, 035702. URL: http://link.aps.org/doi/10.1103/PhysRevLett.114.035702, doi:10.1103/PhysRevLett.114.035702.

[18] Kapfer, S.C., Krauth, W., 2016. Cell-veto Monte Carlo algorithm for long-range systems. Phys. Rev. E 94, 031302. URL: http://link.aps.org/doi/10.1103/PhysRevE.94.031302, doi:10.1103/PhysRevE.94.031302.

[19] Kapfer, S.C., Krauth, W., 2017. Irreversible Local Markov Chains with Rapid Convergence towards Equilibrium. Phys. Rev. Lett. 119, 240603. URL: https://link.aps.org/doi/10.1103/PhysRevLett.119.240603, doi:10.1103/PhysRevLett.119.240603.

[20] Khan, M.A., Herbordt, M.C., 2011. Parallel discrete molecular dynamics simulation with speculation and in-order commitment. Journal of Computational Physics 230, 6563 – 6582. URL: http://www.sciencedirect.com/science/article/pii/S0021999111002968, doi:https://doi.org/10.1016/j.jcp.2011.05.001.

[21] de Leeuw, S.W., Perram, J.W., Smith, E.R., 1980. Simulation of electrostatic systems in periodic boundary conditions. II. Equivalence of boundary conditions. Proc. R. Soc. A 373, 57–66. URL: http://rspa.royalsocietypublishing.org/content/373/1752/57, doi:10.1098/rspa.1980.0136.

[22] Lei, Z., Krauth, W., 2018a. Irreversible Markov chains in spin models: Topological excitations. EPL 121, 10008.

[23] Lei, Z., Krauth, W., 2018b. Mixing and perfect sampling in one-dimensional particle systems. EPL 124, 20003. URL: http://stacks.iop.org/0295-5075/124/i=2/a=20003.

[24] Lei, Z., Krauth, W., Maggs, A.C., 2019. Event-chain Monte Carlo with factor fields. Physical Review E 99. URL: https://doi.org/10.1103/physreve.99.043301, doi:10.1103/physreve.99.043301.

[25] McKerns, M.M., Aivazis, M.A., 2010. Pathos: a framework for heterogeneous computing. URL: http://trac.mystic.cacr.caltech.edu/project/pathos.

[26] McKerns, M.M., Strand, L., Sullivan, T., Fang, A., Aivazis, M.A., 2011. Building a Framework for Predictive Science , in: van der Walt, S., Millman, J. (Eds.), Proceedings of the 10th Python in Science Conference, pp. 67 – 78.

[27] Michaud-Agrawal, N., Denning, E.J., Woolf, T.B., Beckstein, O., 2011. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. Journal of Computational Chemistry 32, 2319–2327. URL: https://doi.org/10.1002/jcc.21787, doi:10.1002/jcc.21787.

[28] Michel, M., Kapfer, S.C., Krauth, W., 2014. Generalized event-chain Monte Carlo: Constructing rejection-free global-balance algorithms from infinitesimal steps. J. Chem. Phys. 140, 054116. doi:10.1063/1.4863991, arXiv:1309.7748.

[29] Miller, S., Luding, S., 2003. Event-driven molecular dynamics in parallel. Journal of Computational Physics 193, 306–316. URL: https://doi.org/10.1016/j.jcp.2003.08.009, doi:10.1016/j.jcp.2003.08.009.

[30] Nishikawa, Y., Michel, M., Krauth, W., Hukushima, K., 2015. Event-chain algorithm for the Heisenberg model: Evidence for z sim 1 dynamic scaling. Phys. Rev. E 92, 063306. doi:10.1103/PhysRevE.92.063306, arXiv:1508.05661.

[31] Oliphant, T.E., 2006. A guide to NumPy. Trelgol Publishing USA.

[32] Peters, E.A.J.F., de With, G., 2012. Rejection-free Monte Carlo sampling for general potentials. Phys. Rev. E 85, 026703. URL: http://link.aps.org/doi/10.1103/PhysRevE.85.026703, doi:10.1103/PhysRevE.85.026703.

[33] Proctor, E.A., Ding, F., Dokholyan, N.V., 2011. Discrete molecular dynamics. Wiley Interdisciplinary Reviews: Computational Molecular Science 1, 80–92. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.4, doi:10.1002/wcms.4, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcms.4.

[34] Proctor, E.A., Dokholyan, N.V., 2016. Applications of Discrete Molecular Dynamics in biology and medicine. Current Opinion in Structural Biology 37, 9 – 13. URL: http://www.sciencedirect.com/science/article/pii/S0959440X15001578, doi:https://doi.org/10.1016/j.sbi.2015.11.001. theory and simulation - Macromolcular machines.

[35] Qin, L., Höllmer, P., Maggs, A.C., Krauth, W., 2019. Benchmarking ECMC against Lammps. Manuscript in preparation.

[36] Shirvanyants, D., Ding, F., Tsao, D., Ramachandran, S., Dokholyan, N.V., 2012. Discrete Molecular Dynamics: An Efficient And Versatile Simulation Method For Fine Protein Characterization. The Journal of Physical Chemistry B 116, 8375–8382. URL: https://doi.org/10.1021/jp2114576, doi:10.1021/jp2114576, arXiv:https://doi.org/10.1021/jp2114576. pMID: 22280505.

[37] Walker, A.J., 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. ACM Trans. Math. Softw. 3, 253–256. URL: http://doi.acm.org/10.1145/355744.355749, doi:10.1145/355744.355749.

[38] van der Walt, S., Colbert, S.C., Varoquaux, G., 2011. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering 13, 22–30. URL: https://doi.org/10.1109/MCSE.2011.37, doi:10.1109/MCSE.2011.37.

[39] Wu, Y., Tepper, H.L., Voth, G.A., 2006. Flexible simple point-charge water model with improved liquid-state properties. The Journal of Chemical Physics 124, 024503. URL: https://doi.org/10.1063/1.2136877, doi:10.1063/1.2136877.

List of Figures