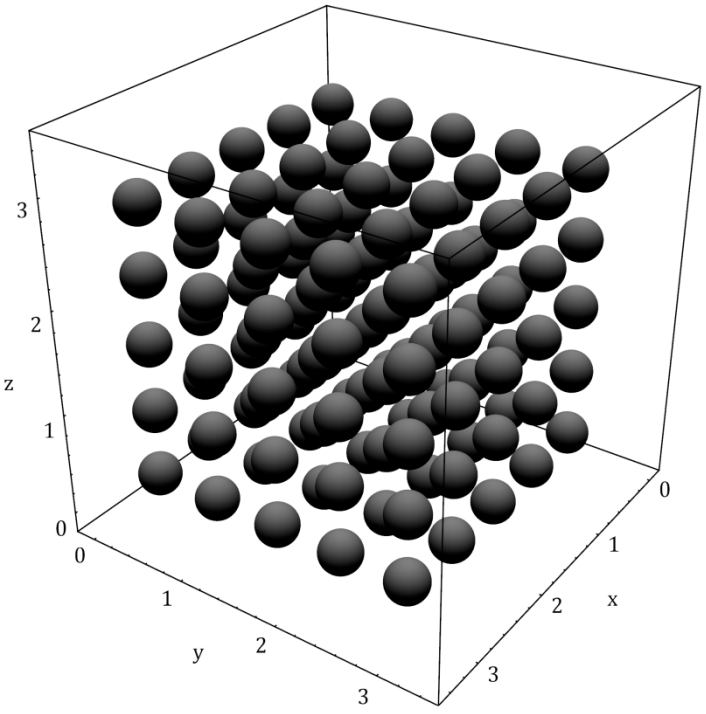
CHALMERS




# Dissipative Particle Dynamics and Coarse-Graining

## Review of Existing Techniques, Trials with Evolutionary Computation

Master's Thesis

ATILIM GÜNEŞ BAYDİN

*Complex Adaptive Systems Programme*
Department of Applied Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2008

Master's Thesis in the Complex Adaptive Systems Programme

# Dissipative Particle Dynamics and Coarse-Graining

Review of Existing Techniques, Trials with Evolutionary Computation

ATILIM GÜNEŞ BAYDİN

Department of Applied Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2008

Dissipative Particle Dynamics and Coarse-Graining
Review of Existing Techniques, Trials with Evolutionary Algorithms
Master's Thesis in the Complex Adaptive Systems Programme
ATILIM GÜNEŞ BAYDİN
Supervised by MARTIN NILSSON JACOBI



Department of Applied Physics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000
Web address: http://www.chalmers.se/ap

Cover:
125 particles arranged in a cubic lattice.

Printed at Chalmers Reproservice
Göteborg, Sweden 2008

Dissipative Particle Dynamics and Coarse-Graining
Review of Existing Techniques, Trials with Evolutionary Computation
Master's Thesis in the Complex Adaptive Systems Programme

ATILIM GÜNEŞ BAYDİN

Department of Applied Physics
Chalmers University of Technology

## ABSTRACT

This thesis provides a review of the dissipative particle dynamics (DPD) technique, a commonly used mesoscopic simulation tool in computational physics; and an investigation of the feasibility of using evolutionary optimization techniques for the determination of interactions in the DPD model from measurements in atomistic simulations. The text starts with a brief overview of the historical development of particle models to provide a foundation for the discussion of coarse-graining, i.e. the description of a system at a less detailed level by smoothing out fine details that are not relevant for a particular study. Detailed introductions of fundamental computational physics methods are presented, such as molecular dynamics and Monte Carlo simulations, together with their application areas. The DPD technique is introduced, with detailed information about its historical development, interpretation as a mesoscopic model, and application areas. The two parts of the DPD coarse-graining process, i.e. the determination of conservative and dissipative interactions, are discussed. Major existing techniques for DPD coarse-graining are presented, such as the inverse Monte Carlo (IMC) procedure specialized for the determination of conservative interactions from structural observables. The thesis continues with an investigation of the feasibility of using evolutionary computation, a generic optimization approach with its roots in the biological process of evolution, for the determination of interactions in the DPD model, based on fitness measures comparing equilibrium and transport properties of the system with those measured in atomistic simulations. Taking the simple point charge water model as a case study, the technique is first used for the determination of conservative interactions from the radial distribution function (with the aim of validating the approach by results from the IMC technique) and after that, for the determination of dissipative interactions based on escape time distributions. The practicality of having relatively long DPD simulations within fitness evaluations of such a procedure is confirmed, also establishing a general framework for applying evolutionary optimization techniques for the determination of functional forms in possibly other models within the field of computational physics.

Keywords: Dissipative particle dynamics; Coarse-graining; Genetic algorithms; Molecular dynamics, Monte Carlo simulations

Dissipativ partikel dynamik och grovkorning
Översyn av befintliga tekniker, prövningar med evolutionära algoritmer
Examensarbete inom masterprogrammet Komplexa adaptiva system

ATILIM GÜNEŞ BAYDİN

Institutionen för teknisk fysik
Chalmers tekniska högskola

## SAMMANFATTNING

Den här avhandlingen ger en översikt över den dissipativ partikel dynamik (DPD) metoden, som är ett vanligt verktyg för mesoskopisk simulering i beräkningsfysik, och en utredning av möjligheterna att använda evolutionära optimering tekniker för bestämning av interaktioner i DPD modellen från mätningar i atomistiska simuleringar. Texten inleds med en kort översyn av den historiska utvecklingen av partikel modeller för att ge en grund för diskussionen av grovkorning, dvs en beskrivning av ett system på en mindre detaljerad nivå genom att jämna ut fina detaljer som inte är relevanta för en särskild studie. Detaljerade introduktioner av grundläggande beräkningsfysik metoder presenteras, som molekylär dynamik och Monte Carlo simuleringar, tillsammans med deras tillämpningsområden. Den DPD teknik införs, med detaljerad information om dess historiska utveckling, uppfattning som en mesoskopisk modell, och applikationsområden. De båda delarna av DPD grovkorning, dvs bestämning av konservativa och dissipativa interaktioner, diskuteras. Viktiga befintliga tekniker för DPD grovkorning presenteras, till exempel invers Monte Carlo (IMC), som är specialiserat för bestämning av konservativa interaktioner från strukturella observabler. Avhandlingen fortsätter med en undersökning av möjligheterna att använda evolutionära beräkning, en generell optimering metod som efterhärmar den biologiska evolutionen, för bestämning av interaktioner i DPD modell, baserad på anpassningsmått som jämför jämviktsläge och transport egenskaper hos systemet med dem som uppmätts i atomistiska simuleringar. Med den enkel punkt laddning vatten modellen som en fallstudie, tekniken används först för bestämning av konservativa interaktioner från radial distribution funktion (i syfte att validera den metoden med resultat från IMC teknik) och sedan, för bestämning av dissipativa interaktioner från flykt tid distributioner. Den möjligheten att ha relativt långa DPD simuleringar inom anpassningsutvärderingar av ett sådant förfarande är bekräftad, även är en allmän inramning bildad för användning av evolutionära optimering tekniker för bestämning av former av funktioner i eventuellt andra modeller inom området beräkningsfysik.

Nyckelord: Dissipativ partikel dynamik; Grovkorning; Genetisk algoritm; Molekylär dynamik, Monte Carlo simulering

# Contents

# Preface

This is the report of the thesis work I have conducted for the completion of my study in the international master's programme of *Complex Adaptive Systems* at the Department of Applied Physics at Chalmers University of Technology. It was performed under the supervision of *Assistant Prof. Martin Nilsson Jacobi* and with the valuable help of his PhD student *Johan Nyström*, in conjunction with their research in the Complex Systems group, Division of Physical Resource Theory, Department of Energy and Environment.

By means of this study, I have had the opportunity to delve rather deep into a field of research that has interested me for long, namely, computational physics. Considering that my knowledge of this field was rather superficial at the beginning of this work, I feel particularly satisfied that most of the domain now seems accessible to me, after going through a good deal of literature and creating implementations for the specific case of dissipative particle dynamics. I hope this work will be useful for others and of relevance to my future studies.

Atılım Güneş Baydin

Göteborg, Sweden

September 2008

# Acknowledgments

I would like to extend my heartfelt thanks to my parents *Nihayet Bayraktar* and *Süleyman Zafer Baydin*, for raising me free from all the dogmas of the society, for their never-ending support, and basically, for bringing me into existence; my dear friend *Melek Tendürüs*, especially for her help in the planning and proofreading stage of this thesis; and my close friends *Emre Duran* and *Koray Savaş Erer*, for keeping me motivated throughout my studies.

# Notations

| | |
|---|---|
| $a$ | Scalar variable |
| $\boldsymbol{a}$ | Vector variable |
| $\|\boldsymbol{a}\|$ | Vector magnitude |
| $\hat{\boldsymbol{a}}$ | Unit vector |
| $\underline{\boldsymbol{A}}$ | Matrix |
| $\langle \cdots \rangle$ | Ensemble average, or a long time average in a single system |

# 1 Introduction

This introductory chapter starts with a very brief overview of the historical development of particle models. Within the general framework of computational physics, the theory molecular dynamics (MD) simulations is dealt with in detail, providing a foundation for the dissipative particle dynamics (DPD) model to be discussed in Chapter 2, with almost all of the implementational details of the MD technique applying equally well to DPD. The Monte Carlo simulation technique is introduced. A general discussion of the shortcomings of the MD technique and the main incentives for the development of mesoscopic simulation techniques are also covered.

## 1.1 The idea of particles

The idea that matter is composed of very small building blocks is traced back to pre-Socratic philosopher Leucippus and his student Democritus(Weinberg, 2003). In his writings in the 5th century BC, Democritus calls these elements ατομα (*atoma,* Greek for "indivisible"; singular: ατομον, *atomon*), which is the basis for the modern name *atom*. The school of thought thus founded, atomism, had its followers until it was marginalized by Aristotle during the late 4th century BC, who maintained that matter is not made of atoms, but rather is continuous. Like with many other fields of physical science, the ideas and influence of Aristotle on this subject lasted all the way until the Renaissance.

During the 17th century, the atomist thought underwent a revival, most notably by the works of Galileo Galilei and René Descartes. The development of modern chemistry provided further support for the idea of atoms, particularly by the work of John Dalton in early the 19th century, using the concept for his explanation of the fact that elements always react in ratios of small natural numbers.

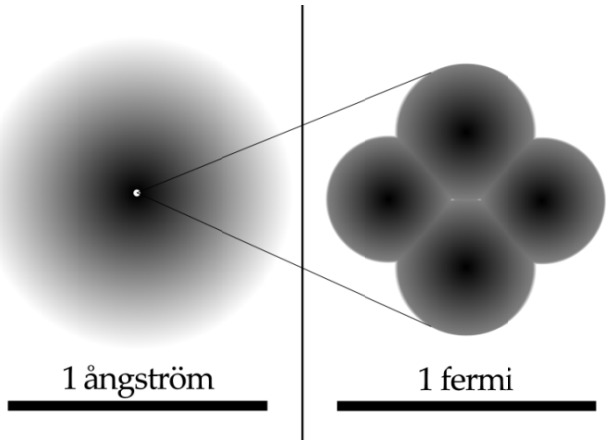



*Figure 1.1 The size of atoms[1] is of the order of* $10^{-10}$ *m (also called* 1 *ångström, named after Anders Jonas Ångström), while the size of atomic nuclei is of the order of* $10^{-15}$ *m (also called* 1 *fermi, after Enrico Fermi). Shown here is the structure of a Helium atom.*

[1] Roughly, the average effective size of the outermost filled electron orbit, as one cannot speak about a definite clear-cut size of an atom.

It was realized during the late 19th and early 20th centuries that atoms are in fact not indivisible, beginning with the discovery of the electron by Joseph John Thomson and subsequent experiments by Ernest Rutherford, which proved that the atom is a largely empty structure with almost all of its mass concentrated in a tiny nucleus orbited by electrons (Figure 1.1) (Veltman, 2003). These developments led to the Bohr model of the atom created by Niels Bohr in 1913, i.e. electrons in orbits of different energy levels around a nucleus composed by protons and neutrons, which still remains the model of atomic structure most commonly known by the general public.

During the 20th century, the trend of discovering further substructures has continued, revealing the divisibility of particles which were previously thought to be indivisible and elementary. For instance, in the current state of quantum physics, protons and neutrons are merely two members within the huge family of particles called hadrons, and are actually composite entities consisting of elementary particles called quarks. Particularly during the 1960s, the discovery of an ever increasing number of particles led to a situation dubbed "the particle zoo", lasting until the formulation of the Standard Model of elementary particles (Figure 1.2Figure 1.2), which postulates two fundamental particle groups: fermions (matter constituents; categorized into quarks and leptons, of which the electron is a type) and bosons (force carriers). Although this current model is in very good agreement with experimental data, it is known to have several shortcomings and expected to get modified, for reasons beyond the scope of this study.

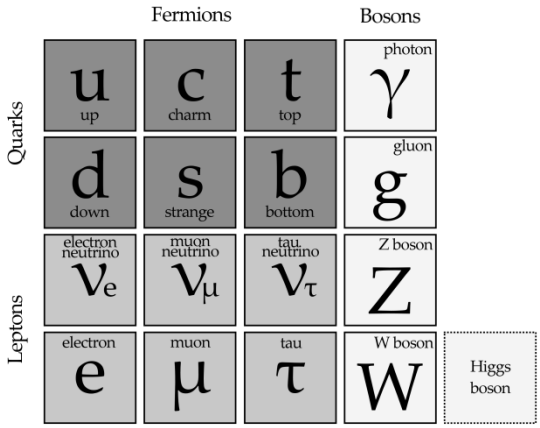



*Figure 1.2 Elementary particles of the Standard Model. The antiparticle counterparts are not shown.*

## 1.2 Computational physics

Rapid advancements in computer science during the late 20th century have made the numerical simulation of complex mathematical models in many fields of science feasible and commonplace. Computer simulations are now a standard tool for experimenting with models not only in fields of natural science as diverse as physics, biology, and earth sciences, but also in social sciences like economics and psychology. Amid these, computational physics has enjoyed particularly success, given that physics as a field features very precise theoretical models which allow calculations replicating real world dynamics to a very high degree of fidelity. In this respect, computational physics is often regarded to fall within the domain of theoretical physics, but it is also possible to consider it a branch of experimental

physics, because it is concerned with the observation of physical phenomena and gathering data.

In this thesis we will focus on computational physics simulations of particle systems. With such simulations, arguably the most important preparation step is to determine the physical scale of the phenomenon that will be the subject of the study and thus the appropriate model providing the right amount of detail.

At the lowest scale, there are what are called a*b initio* (or *first principles*) computer simulation techniques, deriving the interactions within the system from quantum mechanical relations, such as the computation of the forces acting on atomic nuclei by solving the electronic structure problem "on-the-fly", i.e. continuously at each time step of the simulation (Marx & Hutter, 2000). Techniques within this family are very accurate and in widespread use in materials science and chemistry, but suffer from extremely high computational costs.

For the purposes of this study, we are concerned with a description of matter at a higher and far less detailed level, without dealing with the degrees of freedom on the quantum scale. In the simulation techniques that will be presented in the following sections, atoms, atom groups within molecules (such as a methyl group $-CH_3$ within an organic chain), or entire molecules can be represented by single point particles which are then governed only by classical mechanics. In these techniques, the interactions within the system are approximated in terms of classical potential functions, which can have two-body, three-body, or many-body forms.

This is also the current first stage of what is called *coarse-graining*, i.e. the description of a system at a less detailed level by smoothing out fine details that are not relevant for a particular study, in order to keep the computational costs at a manageable level. The subject of coarse-graining will be covered in detail in Chapter 3.

### 1.2.1 Molecular dynamics (MD)

Molecular dynamics (MD) is a simulation technique in which the dynamics of a system of particles is obtained by the numerical integration of their equations of motion under classical (or Newtonian) mechanics, developed during the 1950s and 1960s by the seminal papers of Alder and Wainwright (1957); Gibson, Goland, Milgram, and Vineyard (1960); and Rahman (1964). The time evolution for a $N$-body MD system is described by the equations of motion

$$\begin{aligned} \frac{\partial \boldsymbol{r}_i}{\partial t} &= \boldsymbol{v}_i \,, \\ \frac{\partial \boldsymbol{v}_i}{\partial t} &= \boldsymbol{a}_i \,, \end{aligned} \tag{1.1}$$

where $\boldsymbol{r}_i$, $\boldsymbol{v}_i$, and $\boldsymbol{a}_i$ are the position, velocity, and acceleration vectors of the $i$th particle, in that order. The accelerations $\boldsymbol{a}_i$ in the system are given by Newton's second law of motion

$$\boldsymbol{F}_i = m_i \boldsymbol{a}_i \,, \tag{1.2}$$

where $m_i$ are particle masses; and the forces $\boldsymbol{F}_i$ are given by the gradients of potential $V$ with respect to the positions of the particles:

$$\boldsymbol{F}_i = -\nabla_{\boldsymbol{r}_i} m_i V(\boldsymbol{r}_i, \dots, \boldsymbol{r}_N)\,. \tag{1.3}$$

The function $V$ represents the potential energy of the particle system for every possible configuration of particles $\{\boldsymbol{r}_i, \dots, \boldsymbol{r}_N\}$ and is commonly defined in terms of the sum of two-body (or pairwise) interactions, although many-body forms are also used in particular areas of research, such as with metals and semiconductors (Carlsson, 1990). $V$ can be written as a sum of pairwise interactions $\phi(r_{ij})$ as

$$V(\boldsymbol{r}_i, \dots, \boldsymbol{r}_N) = \sum_i \sum_{j>i} \phi(r_{ij})\,, \tag{1.4}$$

where $r_{ij} = |\boldsymbol{r}_i - \boldsymbol{r}_j|$ is the distance between the pair of particles $i$ and $j$; and the condition $j > i$ in the inner summation ensures that the contribution of each pair to the potential is only considered once.

A common example of pair potentials is the Lennard-Jones potential (named after John Edward Lennard-Jones, who formulated it in 1924), basically describing atoms of a noble gas interacting at long range through attracting van der Waals forces, and at short range, repelling forces resulting from overlapping electron orbits (Figure 1.3). It is given by the formula:

$$\phi_{LJ}(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{1.5}$$

where $\epsilon$ determines the depth of the attractive potential well and $\sigma$ determines the distance where the potential crosses zero and continues to positive infinity, effectively determining the radius of particles in the system. The Lennard-Jones potential was used in earliest MD simulations of liquid argon (Rahman, 1964; Verlet, 1967) and is still commonly encountered in recent studies.

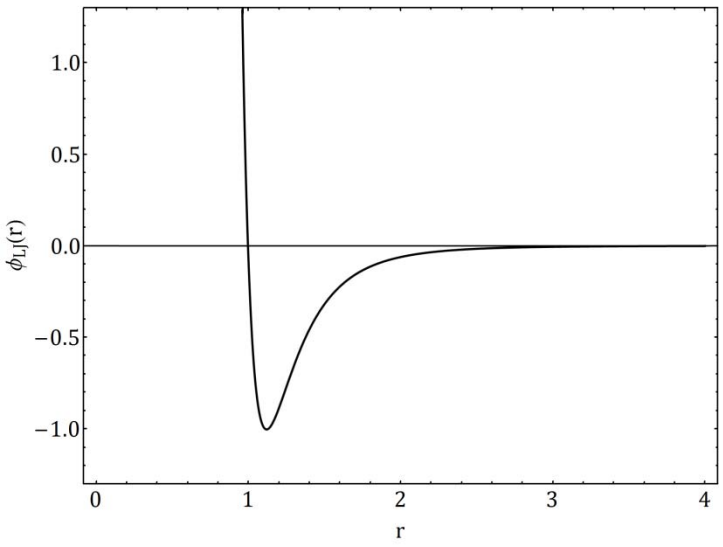



*Figure 1.3 The Lennard-Jones potential with $\varepsilon = \sigma = 1$.*

An important part of MD simulations is the selection of a time integration algorithm that will be used to integrate the equations of motion (1.1) and produce the system dynamics. These use the *finite difference method* of approximating solutions of differential equations, using discretized time with a small time step $\Delta t$, and as such, are prone to the accumulation of truncation and round-off errors during the run of simulations. Since the MD technique is used to compute the time evolution of a system over a very large number of time steps, a simple Euler type integrator

$$\begin{aligned} \boldsymbol{r}_i(t+\Delta t) &= \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2 \,, \\ \boldsymbol{v}_i(t+\Delta t) &= \boldsymbol{v}_i(t) + \boldsymbol{a}_i(t)\Delta t \,, \\ \boldsymbol{a}_i(t+\Delta t) &= \frac{\boldsymbol{F}_i(t+\Delta t)}{m_i} \end{aligned} \tag{1.6}$$

produces numerical errors ($O(\Delta t^3)$ for $\boldsymbol{r}_i$ and $O(\Delta t^2)$ for $\boldsymbol{v}_i$) that are too big to tolerate (Giordano & Nakanishi, 2006).

To reduce the errors associated with time integration, a large variety of integration schemes has been developed over the years and the scheme known as the *Verlet algorithm* (Verlet, 1967) is frequently encountered. For performing the simulations presented in the following chapters of this study, a variation of this standard algorithm, called the *velocity Verlet algorithm* (Swope, Andersen, Berens, & Wilson, 1982) (equations (1.7)) has been implemented. Derivations of the standard and velocity Verlet algorithms are given in Appendix A.

$$\begin{aligned} \boldsymbol{r}_i(t+\Delta t) &= \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2 \\ \boldsymbol{v}_i(t+\frac{\Delta t}{2}) &= \boldsymbol{v}_i(t) + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t \\ \boldsymbol{a}_i(t+\Delta t) &= \frac{\boldsymbol{F}_i(t+\Delta t)}{m_i} \\ \boldsymbol{v}_i(t+\Delta t) &= \boldsymbol{v}_i(t+\frac{\Delta t}{2}) + \frac{1}{2}\boldsymbol{a}_i(t+\Delta t)\Delta t \end{aligned} \tag{1.7}$$

When putting the time integration algorithm into practice, the interaction potential (which, by definition, has infinite range but approaches zero at long distances) is nearly always truncated at a carefully selected *cutoff distance* $r_c$ to achieve generous savings of computational cost, discarding a large number of potential evaluations that would contribute very little to the potential. To avoid artifacts in the conservation of energy, i.e. energy jumps induced by pairs crossing over the cutoff distance, the value of the interaction potential at $r_c$ should be very close to zero. As an example, for the pairwise Lennard-Jones potential $\phi_{LJ}(r)$, this can be assured by shifting the potential so that it vanishes at $r_c$ (equation (1.8)). A commonly used value of the cutoff distance for the Lennard-Jones potential is 2.5 $\sigma$ (Figure 1.3).

$$\phi'_{LJ}(r) = \begin{cases} \phi_{LJ}(r) - \phi_{LJ}(r_c) & \text{if } r \le r_c \\ 0 & \text{if } r > r_c \end{cases} \tag{1.8}$$

Another important topic in the implementation of the MD technique, and other particle simulation techniques in general, is how the boundary conditions are handled. Except in studies specifically interested in superficial phenomena like the analysis of surface tension or multi-phase systems, MD simulations are generally performed using *periodic boundary conditions*. This is achieved by defining a simulation box and having all its sides loop back to the opposite side, such that when a particle leaves the simulation box by passing through a side of the simulation box, it will reappear at the opposite side. This has the same effect with envisioning that the simulation box is replicated indefinitely along all Cartesian directions and that each particle in the original box represents an infinite set of its images in each of these boxes, effectively obtaining an infinite system completely filling space. This would mean that particles sufficiently close (less than $r_c$) to a boundary will interact not only with the other particles in the simulation box, but also with their images in the neighboring boxes. A substantial simplification of this complicated interaction scheme is provided by what is called the *minimum image criterion*: If all side lengths of the simulation box are greater than $2r_c$, any neighboring images of a particle $j$ will be separated by more than $2r_c$, meaning that a particle $i$ can be within the cutoff distance $r_c$ with at most one of these images (Figure 1.4). In other words, of all the images of another particle $j$, a particle $i$ will only interact with the closest, on the condition that all side lengths of the simulation box are greater than $2r_c$ (Rapaport, 2004).

Even if the cutoff distance technique significantly improves the computation time spent by the evaluation of potentials in a time step of the simulation, there still remains the necessity to compute the distances between all particle pairs at each time step, to check whether these are less than $r_c$. Another commonly used technique in the implementation of MD simulations, the *Verlet neighbor lists* introduced by Verlet (1967), provides considerable speedups by keeping a list of "neighbors" for each particle, i.e. those that are within a range $r_n > r_c$ of the particle, and using these lists for the evaluation of interactions. This list can updated with an arbitrary interval of time steps determined beforehand(Verlet, 1967), or dynamically when any particle has moved a distance greater than $\frac{1}{2}(r_n - r_c)$ (Chialvo & Debenedetti, 1992), which generally occurs every 10 – 20 time steps with a carefully selected $r_n$.

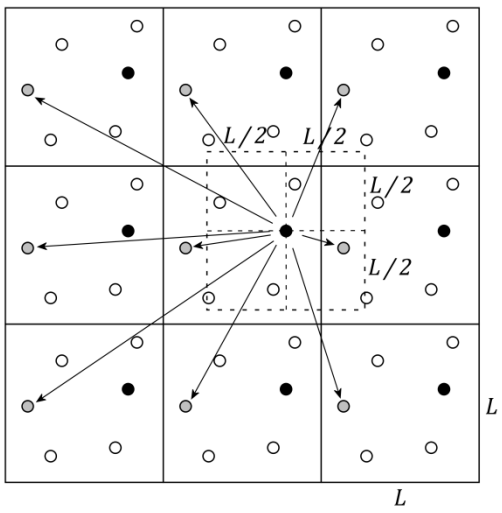



*Figure 1.4 The nearest image criterion.*

The primary observables in classical MD simulations are macroscopic thermodynamic properties such as pressure, energy, heat capacity, etc. The connection between the microscopic measurements on the simulated particles and these macroscopic observables is provided by statistical mechanics. MD simulations generate a sequence of points in the microscopic phase space as a function of time, i.e. any computed averages will be time averages of just one system; while thermodynamic observables are defined in terms of ensemble averages, i.e. averages taken over a large number of replicas of the system. The link between these two is established by the *ergodic hypothesis*, supposing that the average of an observable over time will be the same with its ensemble average:

$$\langle A \rangle_{\text{ensemble}} = \langle A \rangle_{\text{time}} . \tag{1.9}$$

This is based on the assumption that if one allows the system to evolve indefinitely in time, the system will eventually pass through all possible microstates. In practice, this means that the measurements will get more accurate with increasing time steps. In addition, because the simulations are of fixed duration, one must make sure that the simulation is performed over a sufficiently large amount of time steps in order to sample a sufficient amount of the phase space.

The most important limitation of the MD technique is that the high computational costs of doing atomistic (or, as it is usually called, microscopic) simulations severely restrict the spatial and temporal scales accessible by this technique, given the average computational resources available today. The length scales recently accessible by MD simulations have been of the order of $10^{-10}$ m (or 1 Å), simulating time spans of the order of $10^{-9}$ s (or 1 ns) (Sutmann, 2002); while simulations going beyond $10^{-6}$ s (or 1 μs) were made possible by distributed parallel algorithms and specialized computer hardware. This limitation is one of the major motivations for the continuing research on coarse-grained mesoscopic models, like lattice based methods briefly discussed in Section 1.2.3 and the dissipative particle dynamics technique that will be introduced in Chapter 2.

### 1.2.2 Monte Carlo (MC) simulations

The Monte Carlo (MC) method is a statistical approach for finding approximate solutions to problems by means of random sampling. In addition to physics, it is widely applied in other natural sciences, mathematics, engineering, and social sciences (Krauth, 2006).

Although much earlier treatments in the subject exist—most notably in connection with the "Buffon's needle problem"[2], such as by Barbier (1860)—the invention of the modern MC technique is associated with Enrico Fermi, when he was studying the properties of the then newly-discovered neutron in 1930 (Metropolis, 1987). It was further developed during the 1940s by physicists working in the nuclear weapons program of the United States, at the Los Alamos National Laboratory(Ulam, Richtmyer, & von Neumann, 1947). The technique was given its name by Nicholas Metropolis, in reference to the famous casino in Monaco,

[2] The problem is, given a needle of length $l$ dropped on a floor striped with parallel lines $t$ units apart, to find the probability that the needle will land such that it crosses a line. (The answer is $(2l)/(t\pi)$.) The problem is first posed by and named after the 18th century French naturalist Georges-Louis Leclerc, Comte de Buffon.

considering the use of randomness and the repetitive nature of the sampling process (Metropolis & Ulam, The Monte Carlo method, 1949).

A definition given by Anderson (1999) describes the MC technique as "the art of approximating an expectation by the sample mean of a function of simulated random variables". This can be clarified by means of a simple example, and a commonly used one is the MC calculation of the value of $\pi$ by simple sampling. Consider the unit square overlapping one quarter of the unit circle in Figure 1.5. If we shoot a number of random points uniformly distributed within the unit square, the ratio of the ones falling within the unit circle to the total number of points will approximate the ratio of the area of the quarter of the unit circle to the area of the unit square, which is exactly $\pi/4$. In Figure 1.5, there are 4000 random points, of which 3152 fall within the unit circle, so that $\pi/4 = 3152/4000$, giving the value of $\pi \cong 3.152$. The approximation will get more accurate with larger numbers of random samples and in the limit of infinity it will be exactly equal to $\pi$ by definition.

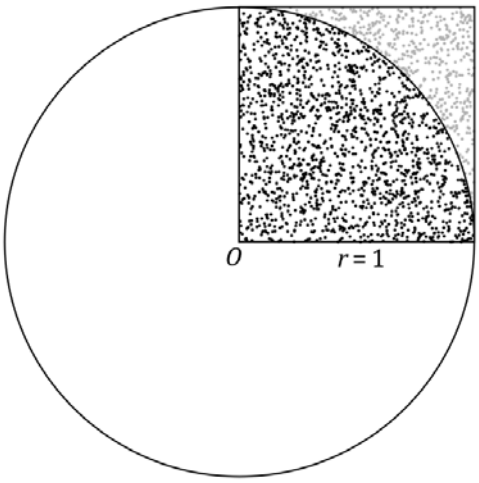



*Figure 1.5 Calculation of the value of π by the Monte Carlo technique.*

Beyond simple sampling with uniformly distributed samples, there is usually a need to cover the sample space according to a specific probability distribution function and this is called importance sampling. A frequently used importance sampling algorithm is the Metropolis algorithm, originally published for the specific case of Boltzmann distribution (Metropolis, Rosenbluth, Rosenbluth, Teller, & Teller, 1953) and later generalized to other distributions (Hastings, 1970). The standard Metropolis algorithm works by constructing a Markov chain that has the Boltzmann distribution as its equilibrium distribution, as follows:

- Generate the initial microstate $S$ and calculate the resulting potential $V = V(S)$
- Modify the microstate $S$ to $S_{\text{test}}$ and calculate the test potential $V_{\text{test}} = V(S_{\text{test}})$
- If $V_{test} < V$ then
    - $S \leftarrow S_{\text{test}}$ and $V \leftarrow V_{\text{test}}$
- else
    - Generate uniform random number $0 \leq R \leq 1$

    - If $e^{-\frac{(V_{\text{test}}-V)}{k_B T}} > R$ ($k_B$: Boltzmann constant; T: temperature) then
        - $S \leftarrow S_{\text{test}}$ and $V \leftarrow V_{\text{test}}$
    - End if
- End if
- Repeat from the second step until the end condition is satisfied

For a molecular MC simulation simulating a system of $N$ particles governed by a potential $V(\boldsymbol{r}_i, \dots, \boldsymbol{r}_N)$, as in equation (1.3), the structure $R$ would comprise the positions of particles $\{\boldsymbol{r}_i, \dots, \boldsymbol{r}_N\}$.

It is important to note that the simulation steps in the MC technique are steps in configuration space and there is no notion of "time" in MC simulations. This is in contrast to MD, where the simulation steps are explicit time steps. Nevertheless, MC is often used to compute time averages in simulated processes and this is again granted by the ergodic hypothesis discussed in the previous section (equation (1.9)), this time in the other direction, assuming that phase space averages are identical to the time averages.

In addition to the classical MC simulations where the Metropolis algorithm with the Boltzmann distribution is used to obtain thermodynamic equilibrium properties, the technique is being used for other studies such as quantum MC (QMC) for solving electronic structure properties and kinetic MC (KMC) for simulating the time evolution of natural processes with known rates.

The MC technique (with the Metropolis algorithm) forms the basis of the inverse Monte Carlo (IMC) technique of reconstructing effective conservative potentials for coarse-grained simulations, which will be discussed in Chapter 3.

### 1.2.3 Other simulation techniques

MD is a very precise simulation technique. However, as discussed in Section 1.2.1, this comes at the cost of high computational requirements and applicability in a narrow spatiotemporal scale, effectively limited to microscopic[3] simulations. When there is a need to simulate phenomena on larger scales, particularly when one is concerned with time scales more relevant to biological processes—such as the formation of lipid membranes or proteins—the so-called mesoscopic models are employed, which reside a few steps over the microscopic, but below the macroscopic, models in the coarse-graining ladder.

An important class of mesoscopic simulations are the lattice gas automata (LGA), which involve the discretization of space as a lattice and the movement of particles according to local rules (Figure 1.6), capable of replicating the macroscopic behavior expected by Navier-Stokes equations of fluid dynamics(Frisch, Hasslacher, & Pomeau, 1986). The relatively new lattice Boltzmann methods (LBM) have been developed from the LGA technique, solving some of the problems intrinsic in LGA, such as high numerical noise (Succi, 2001).

[3] In these types of studies, "microscopic" is usually meant to denote all-atom (atomistic) simulations close to nm scale, rather than the conventional meaning of the word denoting objects that can be seen under light microscope.

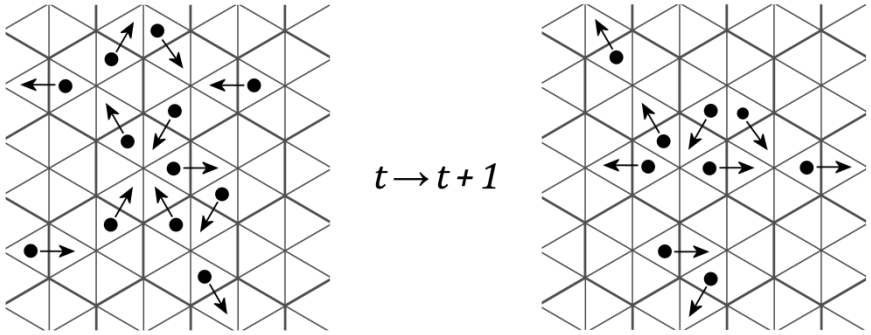



*Figure 1.6 A lattice gas automaton on hexagonal grid.*

Even if lattice methods like LGA and LBM have major advantages in dealing with complex boundaries, computations without any round-off error and absolute stability, and being very suitable for implementations on parallel computer architectures, they have also serious drawbacks such as the lack of Galilean invariance and the difficulty of handling three dimensional problems. Another family of mesoscopic techniques has been created as a result of similar considerations that led to the development of lattice methods, while largely alleviating their shortcomings. These are based on stochastic Langevin equations, smoothing out fast degrees of freedom within the system, such as Brownian dynamics (BD) and its close relative dissipative particle dynamics (DPD), which will be introduced in Chapter 2 as the main subject of this thesis.

For the simulation of macroscopic scales, various specialized techniques such as vortex methods (VM) and smoothed particle hydrodynamics (SPH) exist, but these are outside the scope of this thesis.

### 1.2.4 Applications

Computational physics simulations today are an essential tool for research in studies such as of liquids, surfaces, biomolecules, plasmas, and solid state phenomena. Of these, molecular biology simulations have been receiving increasing attention, such as protein structure prediction and the simulation of their interactions with other molecules (Figure 1.7). At the same time, attempts of simulating systems of very large size are being made, like the recent 50 ns MD simulation of the complete *Satellite Tobacco Mosaic Virus* comprising one million atoms (Freddolino, Arkhipov, Larson, McPherson, & Schulten, 2006). One particularly important area receiving substantial interest and funding is the pharmaceutical research on biomolecules and drug design. Computational studies on the delivery, action mechanisms, and metabolization of drugs allow very early testing of their properties, cutting costs by eliminating the need to synthesize the actual drug for laboratory tests until a far later stage of research.

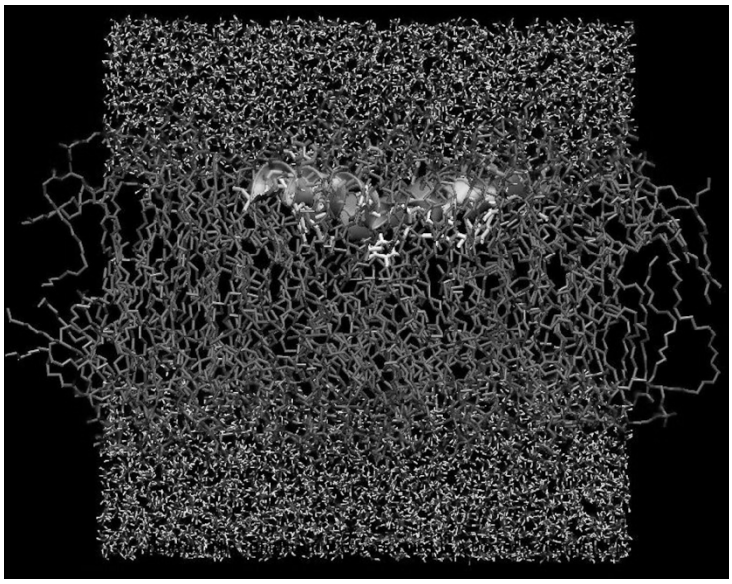


*Figure 1.7 Snapshot of a molecular dynamics simulation of the pore-forming peptide maganin, in a lipid bilayer (Dodd & Dempsey, 2008).*

## 1.3 Organization of the thesis

The remaining part of the thesis after this introduction is organized as follows. In Chapter 2, the dissipative particle dynamics (DPD) simulation technique, commonly interpreted as simulating soft matter systems in mesoscopic scales, is introduced in detail; followed in Chapter 3 by a discussion of the coarse-graining process and the techniques through which the interactions in the DPD model are derived from microscopic simulations, such as the inverse Monte Carlo (IMC) procedure. Chapter 4 presents tests evaluating the feasibility of using evolutionary algorithms (specifically, genetic algorithms) in deriving the form of coarse-grained forces in the DPD model and how these results compare to existing coarse-graining techniques. The thesis is concluded in Chapter 5. In the appendices that follow, derivations of the used time integration algorithms, together with the source code of the implementations written for this thesis, are given.

# 2 Dissipative Particle Dynamics (DPD)

Molecular dynamics, covered in the previous chapter, is a powerful simulation technique proven to produce highly realistic results in a wide variety of applications. However, the computational costs of the detailed interaction model in this paradigm severely limit its applicability beyond extremely small spatiotemporal scales. Within the family of simulation techniques designed to overcome this limitation, we focus on dissipative particle dynamics, which allows, at the cost of reduced resolution, the study of complex hydrodynamic phenomena in extensive scales. This chapter introduces the technique through following its chronological development, also discussing its applications and how it compares to other particle simulation techniques in terms of scale and performance.

## 2.1 Original formulation by Hoogerbrugge and Koelman

The dissipative particle dynamics (DPD) technique has been introduced in the 1990s as a novel scheme for mesoscopic simulations of complex fluids (Hoogerbrugge & Koelman, 1992; Koelman & Hoogerbrugge, 1993). In addition to interactions based on a conservative potential like in a MD simulation, it includes dissipative and stochastic interactions. This approach is ultimately based on the Langevin equation, a stochastic differential equation describing Brownian motion in a potential, accounting for the omitted degrees of freedom by a viscous force and a noise term.

The original DPD model is described by

$$\begin{aligned} \frac{\partial \boldsymbol{r}_i}{\partial t} &= \boldsymbol{v}_i \,, \\ m_i \frac{\partial \boldsymbol{v}_i}{\partial t} &= \boldsymbol{F}_i \,, \end{aligned} \tag{2.1}$$

where $\boldsymbol{r}_i$, $\boldsymbol{v}_i$, and $m_i$ are the position, velocity, and mass of particle $i$, respectively. The total force $\boldsymbol{F}_i$ acting on each particle consists of three parts:

$$\boldsymbol{F}_i = \sum_{j \neq i} \left( \boldsymbol{F}_{ij}^C + \boldsymbol{F}_{ij}^D + \boldsymbol{F}_{ij}^S \right) , \tag{2.2}$$

where $\boldsymbol{F}_{ij}^C$, $\boldsymbol{F}_{ij}^D$, and $\boldsymbol{F}_{ij}^S$ represent the conservative, dissipative, and stochastic forces between particles $i$ and $j$, in that order. The conservative force

$$\boldsymbol{F}_{ij}^C = F^C\left(r_{ij}\right) \hat{\boldsymbol{r}}_{ij} \,, \tag{2.3}$$

where $F^C(r)$ is a non-negative (repulsive) scalar function determining the form of conservative interactions, $\boldsymbol{r}_{ij} = \boldsymbol{r}_i - \boldsymbol{r}_j$ is the distance between particles $i$ and $j$, $r_{ij} = \left\| \boldsymbol{r}_{ij} \right\|$ is its magnitude, and $\hat{\boldsymbol{r}}_{ij} = \boldsymbol{r}_{ij} / r_{ij}$ is the unit vector from $j$ to $i$, depends on the particular

system of interest, although in literature it is frequently taken as a soft repulsion with the form

$$F^C(r) = \begin{cases} a\left(1 - \frac{r}{r_c}\right) & \text{if } r \leq r_c \\ 0 & \text{if } r > r_c \end{cases}, \tag{2.4}$$

where $a$ is a parameter determining the maximum repulsion between the particles and $r_c$ is the cut-off distance.

The dissipative and stochastic forces are given by

$$\begin{aligned} \boldsymbol{F}_{ij}^D &= -\gamma\omega(r_{ij})(\hat{\boldsymbol{r}}_{ij} \cdot \boldsymbol{v}_{ij})\hat{\boldsymbol{r}}_{ij}\,, \\ \boldsymbol{F}_{ij}^S &= \sigma\omega(r_{ij})\zeta_{ij}\hat{\boldsymbol{r}}_{ij}\,, \end{aligned} \tag{2.5}$$

where $\gamma$ and $\sigma$ are parameters determining the strength of dissipative and stochastic interactions, $\omega(r)$ is a non-negative weight function described further below, and $\boldsymbol{v}_{ij} = \boldsymbol{v}_i - \boldsymbol{v}_j$ is the difference in the velocities of particles $i$ and $j$. $\zeta_{ij}$ is a Gaussian white-noise term with the properties

$$\begin{aligned} &\zeta_{ij}(t) = \zeta_{ji}(t)\,, \\ &\langle\zeta_{ij}(t)\rangle = 0\,, \\ &\langle\zeta_{ij}(t)\zeta_{i'j'}(t')\rangle = \left(\delta_{ii'}\delta_{jj'} + \delta_{ij'}\delta_{ji'}\right)\delta(t - t')\,, \end{aligned} \tag{2.6}$$

where $\delta_{ij}$ is the Kronecker delta function and $\delta(t)$ is the Dirac delta function. The condition of symmetry between $\zeta_{ij}$ and $\zeta_{ji}$ ensures the conservation of momentum by the stochastic force. In practice, $\zeta_{ij}$ is commonly implemented as a uniform random variable, instead of Gaussian, taking less computational time to generate (Groot & Warren, Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation, 1997). It has been customary in literature for the weight function $\omega(r)$ to have the form

$$\omega(r) = \begin{cases} 1 - \frac{r}{r_c} & \text{if } r \leq r_c \\ 0 & \text{if } r > r_c \end{cases}, \tag{2.7}$$

since it was introduced as a simple choice by Hoogerbrugge and Koelman (1992) (Figure 2.1).

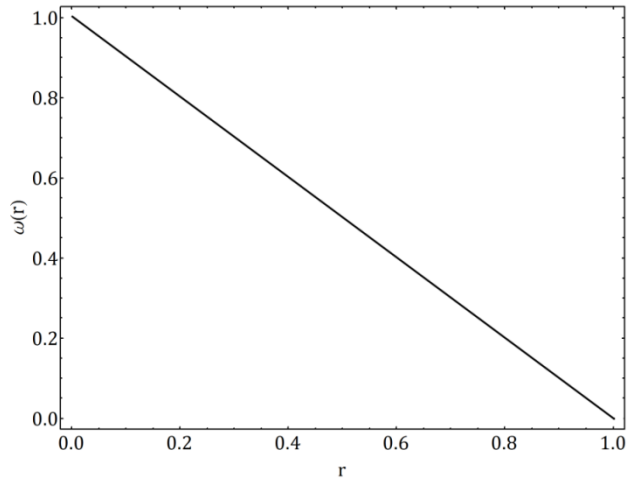



*Figure 2.1 The common form of $\omega(r)$, with $r_c = 1$.*

One important consequence of the DPD formulation is that all interactions are pairwise additive and satisfy Newton's third law, so that the linear and angular momentum is conserved (Hoogerbrugge & Koelman, 1992) and the fact that all the forces depend only on relative positions $\boldsymbol{r}_{ij}$ and velocities $\boldsymbol{v}_{ij}$ makes the model Galilean-invariant. The satisfaction of these conditions make DPD a consistent hydrodynamics model particularly appealing for the study of mesoscopic soft matter systems, so that it has gained substantial support in literature, which led the way for rigorous theoretical analyses of its hydrodynamic and thermodynamic properties and its further development.

## 2.2 Español and Warren's constraint

Despite qualitative observations, there was no theoretical justification that DPD has correct hydrodynamic behavior until the undertaking by Español and Warren (1995), formulating first the Fokker-Planck equation for studying the equilibrium properties of the stochastic differential equation describing DPD; and by Español (1995), deriving the macroscopic hydrodynamic variables starting from a microscopic description.

The main result of these studies is that, unlike the conservative force $\boldsymbol{F}_{ij}^{C}$, the dissipative and stochastic forces $\boldsymbol{F}_{ij}^{D}$ and $\boldsymbol{F}_{ij}^{S}$ cannot be independent and must be coupled together through a fluctuation-dissipation relation, so that equation (2.5) becomes

$$\begin{aligned} \boldsymbol{F}_{ij}^{D} &= -\gamma \omega^{D}(r_{ij})(\hat{\boldsymbol{r}}_{ij} \cdot \boldsymbol{v}_{ij})\hat{\boldsymbol{r}}_{ij} \,, \\ \boldsymbol{F}_{ij}^{S} &= \sigma \omega^{S}(r_{ij})\zeta_{ij}\hat{\boldsymbol{r}}_{ij} \,, \end{aligned} \tag{2.8}$$

with the condition

$$\sigma^2 = 2\gamma k_B T \,, \tag{2.9}$$

where $k_B$ is the Boltzmann's constant and $T$ is the temperature and $\omega^{D}(r)$ and $\omega^{S}(r)$ are separate weight functions satisfying

$$\omega^D(r) = [\omega^S(r)]^2. \tag{2.10}$$

These conditions are necessary to ensure that in thermodynamic equilibrium the dissipative and stochastic forces will keep the system in the canonical, or NVT, ensemble4. For simplicity, the weight functions are usually selected to be similar in form to the conservative force $F^C(r)$ (Figure 2.1), that is

$$\omega^D(r) = [\omega^S(r)]^2 = \begin{cases} \left(1 - \frac{r}{r_c}\right)^2 & \text{if } r \le r_c \\ 0 & \text{if } r > r_c \end{cases}. \tag{2.11}$$

An alternative notation for the formulae describing the dissipative and stochastic part of the dynamics is given by Eriksson, Jacobi, Nyström, and Tunstrøm (2008a), where they combine equations (2.8), (2.9), and (2.10) into the simplified form

$$\begin{aligned} \boldsymbol{F}_{ij}^D &= -\left[\omega(r_{ij})\right]^2 (\hat{\boldsymbol{r}}_{ij} \cdot \boldsymbol{v}_{ij}) \hat{\boldsymbol{r}}_{ij}\,, \\ \boldsymbol{F}_{ij}^S &= \sqrt{2k_B T}\, \omega(r_{ij})\, \zeta_{ij} \hat{\boldsymbol{r}}_{ij}\,, \end{aligned} \tag{2.12}$$

and absorb the parameter $\sigma$ into the definition of the function $\omega(r)$, for instance,

$$\omega(r) = \begin{cases} \sigma\left(1 - \frac{r}{r_c}\right) & \text{if } r \le r_c \\ 0 & \text{if } r > r_c \end{cases}. \tag{2.13}$$

Using this notation, the final description of the DPD dynamics is formed by equations (2.1) – (2.3) and (2.12), where the function $F^C(r)$ describing conservative dynamics and the function $\omega(r)$ describing the dissipative and stochastic dynamics need to be determined for the particular system under study.

## 2.3 System of units

Simulations with the DPD model, as with other models in general, are conventionally performed in non-dimensionalized or reduced units, based on the characteristic physical dimensions of the system under study. Working with reduced units is preferred mainly because they are physically more meaningful and easier to interpret, and the results obtained become applicable to all materials modeled by the same potential, i.e. preventing the conduction of essentially duplicate simulations (Hadjiconstantinou, 2006).

Reduced units are obtained by expressing all the quantities in the simulation in terms of selected base units which are characterizing the system, in order to make the equations

[4] The canonical or NVT ensemble is a statistical ensemble of microstates in which the number of particles N, system volume V, and temperature T are conserved (i.e. the system is in energy exchange with a heat bath). This is in contrast to the natural ensemble of MD simulations, the microcanonical or NVE ensemble, where energy E is constrained and the temperature is free to fluctuate.

dimensionless, such as picking the value of $k_B T$ in Joules (depending on temperature) as a base unit $\varepsilon$ for energy and dividing all quantities with the dimension of energy with $\varepsilon$. Table 2.1 presents further examples.

*Table 2.1 Conversion to and from reduced units for some commonly used dimensions, with $\varepsilon$, $\lambda$, and $\mu$ as the base units for energy, length, and mass, respectively.*

| **Dimension** | **In reduced units** | **In physical units** |
|---|---|---|
| Energy | $E^* = \frac{E}{\varepsilon}$ | $E = \varepsilon E^*$ |
| Length | $L^* = \frac{L}{\lambda}$ | $L = \lambda L^*$ |
| Mass | $m^* = \frac{m}{\mu}$ | $m = \mu m^*$ |
| Temperature | $T^* = \frac{k_B T}{\varepsilon}$ | $T = \frac{\varepsilon T^*}{k_B}$ |
| Density | $\rho^* = \lambda^3 \rho$ | $\rho = \frac{\rho^*}{\lambda^3}$ |
| Force | $F^* = \frac{\lambda F}{\varepsilon}$ | $F = \frac{\varepsilon F^*}{\lambda}$ |
| Pressure | $P^* = \frac{\lambda^3 P}{\varepsilon}$ | $P = \frac{\varepsilon P^*}{\lambda^3}$ |
| Time | $t^* = t\sqrt{\frac{\varepsilon}{\mu\lambda^3}}$ | $t = t^*\sqrt{\frac{\mu\lambda^3}{\varepsilon}}$ |

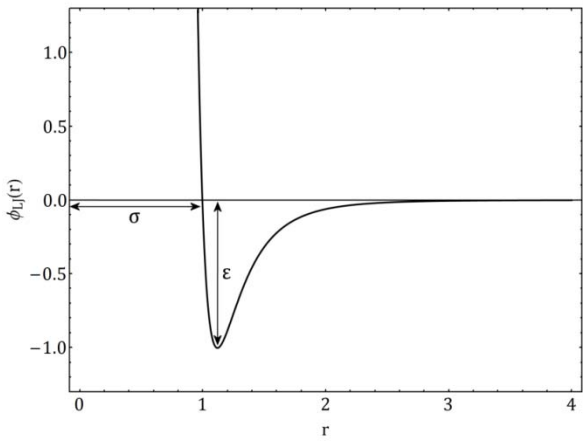



*Figure 2.2 The Lennard-Jones potential and basis for reduced units.*

As an example, for the Lennard-Jones potential (equation (1.5)), the particle diameter $\sigma$ and the depth of the potential well $\varepsilon$, together with the mass of the simulated particles, provide a meaningful set of base units for simulations (Figure 2.2). Another commonly used and convenient basis for the reduced length is the cutoff distance $r_c$.

In computer implementation of simulations, reduced units have the benefit of keeping the computed values within a manageable range (e.g. a length of 0.5 $\sigma$ instead of $5 \times 10^{-10}$ m) so that the risk of arithmetic overflows are eliminated. Another important advantage brought by the use of reduced units is increased computational efficiency in the execution of algorithms, e.g. for the case of Lennard-Jones potential, taking $\varepsilon$ and $\sigma$ as the base units reduces equation (1.5) to the computationally more efficient form:

$$\phi_{LJ}(r) = 4(r^{-12} - r^{-6}) \,. \tag{2.14}$$

All quantities given in the rest of this thesis are in reduced units, unless noted otherwise.

## 2.4 Interpretation and validity

In the seminal paper by Hoogerbrugge and Koelman (1992), the DPD is introduced as a "novel particle-based scheme combining the best of both MD and LGA simulations" which is "much faster than MD and much more flexible than LGA". The interpretation of DPD in the literature has been mostly in line with this, i.e. as a different and complete coarse-grained model by itself, with the form of the conservative and stochastic/dissipative forces depending on a particular system of interest.

However, one should also note that although DPD can be seen as a complete coarse-grained model with the conservative and stochastic/dissipative forces as its integral parts, it may also be treated just as a novel thermostat with the stochastic/dissipative forces, applicable to any MD simulation, which conceptually owns and brings with it the conservative force (Groot & Warren, Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation, 1997; Soddemann, Dünweg, & Kremer, 2003). Viewed in this way, it is comparable to other MD thermostats generating the NVT ensemble, such as the Nosé-Hoover (NH) thermostat (Nosé, 1984; Hoover, 1985) or the stochastic dynamics (SD) thermostat (Schneider & Stoll, 1978), with significant advantages like locality and momentum conservation leading to correct hydrodynamics and the stabilization of the numerical integration scheme (Soddemann et al., 2003).

Another topic of importance regarding DPD is its interpretation as a coarse-grained mesoscopic model. This is again suggested first by Hoogerbrugge and Koelman (1992), stressing that the DPD scheme reproduces the Navier-Stokes behavior "with a number of particles orders of magnitude lower than the number typically needed in LGA and MD simulations", justifying an interpretation that a single DPD particle actually represents a larger number of atoms or molecules that would be represented individually, for instance, in a MD simulation(Español, 1995). This, coupled with the frequent use of soft conservative potentials in DPD simulations—meaning that DPD particles are soft clusters of molecules—established the common understanding of DPD as a mesoscopic particle model. Again, there are different views on the issue and there exist studies employing DPD just as a thermostat

to MD simulations with atomistic, hard particles (Dzwinel & Yuen, 2000; Soddemann, Dünweg, & Kremer, 2003).

There have also been discussions regarding the true scalability of the DPD scheme. Arguments have been raised that the applicability and performance of DPD is dependent upon the level of coarse-graining (Dzwinel & Yuen, 2000) and that there exists a rather low coarse-graining limit above which the technique exhibits unrealistic thermodynamic behavior and ceases to be applicable(Trofimov, 2003). Recently, Füchslin, Fellermann, Eriksson, and Ziock (2007) addressed these concerns by proposing a physically consistent scaling scheme for DPD, asserting that this allows it to function truly scale-free.

## 2.5 Applications

As noted in Chapter 1, the applicability of all-atom simulation techniques such as MD is currently limited to just a few tens of nanoseconds of molecular motion. In fields such as molecular biology and some subfields of chemistry, many interesting phenomena occur in time and length scales much larger than those pertaining to the motion of individual atoms or molecules, such as membrane structuring (Groot, 2004). The DPD technique has been developed in order to allow simulations in scales more relevant to these processes, by omitting degrees of freedom not immediately essential for the description of the system at the studied level. Using mesoscopic descriptions, DPD models allow simulating length scales reaching micrometers, while the use of soft interactions make feasible relatively large integration time steps and total simulation times of the order of microseconds (Sutmann, 2002; Heyes, Baxter, Tüzün, & Qin, 2004).

Because mesoscopic DPD simulations allow access to relatively large length and time scales and creating simulations of multi-component systems with DPD is rather straightforward—by using tags to distinguish particles of a particular type and making the forces in the system dependent on the tags of interacting particles—DPD has been a favored simulation technique for simulations involving a large variety of fields, including fluid mechanics (Kim & Phillips, 2004), colloidal suspensions (Boek, Coveney, & Lekkerkerker, 1996; Whittle & Dickinson, 2001), polymers (Schlijper, Hoogerbrugge, & Manke, 1995; Chen, Phan-Thien, Fan, & Khoo, 2004), pattern formation in developmental biology (Ceicedo-Carvajal & Shinbrot, 2008), medicine (Dzwinel, Boryczko, & Yuen, 2003; Filipovic, Kojic, & Tsuda, 2006), and molecular self-assembly (Nakamura & Tamura, 2005; Wu, Xu, Xianfeng, Yuehong, & Wen, 2006).

### 2.5.1 The PACE project

The Programmable Artificial Cell Evolution (PACE) project[5] aims to lay the theoretical and experimental foundations for creating the first generation of self-assembling, programmable, artificial protocells, i.e. cell-like structures of minimal complexity which are capable of sustaining a simple metabolism along with self-replication (PACE Consortium, 2007). The project is one of the major organized efforts in the direction of producing real *artificial life*—as opposed to ongoing work centered on abstract computer simulations—with the potential of

[5] The PACE project has been carried out since 2004 by a consortium of 13 partners from 8 European countries in cooperation with several organizations from the United States and is funded under the IST-FET (Future and Emerging Technologies) section of the European Union's 6th Framework Programme.

explaining how and under which conditions the self-sustaining and evolving collections of matter that we call "living" arise from "non-living" molecules, one of the yet unanswered fundamental questions in science.

An early concept for a simple protocell is the so-called Los Alamos Bug (Figure 2.3) developed at the Los Alamos National Laboratory (LANL) of the United States. The LANL currently hosts the Protocell Assembly (PAs) project (in collaboration with PACE), the objective of which is defined as "*to produce a minimal self-replicating molecular machine*"(Protocell Assembly Project, 2004).

In the PACE project, the experimental work depends on the subfield of nanotechnology called microfluidics, which allows precise digital control of biochemical reactions; while an important part of the theoretical work is related to DPD simulations of self-assembling micellar systems and mesoscopic models of membrane-metabolism vesicles (Fellermann, Rasmussen, Ziock, & Solé, 2007). Extended DPD models such as the multipole reactive DPD (mprDPD) are being used to study the interaction of reaction-diffusion pattern formation with self-assembled mesoscale structures (Füchslin, Maeke, & McCaskill, 2007).

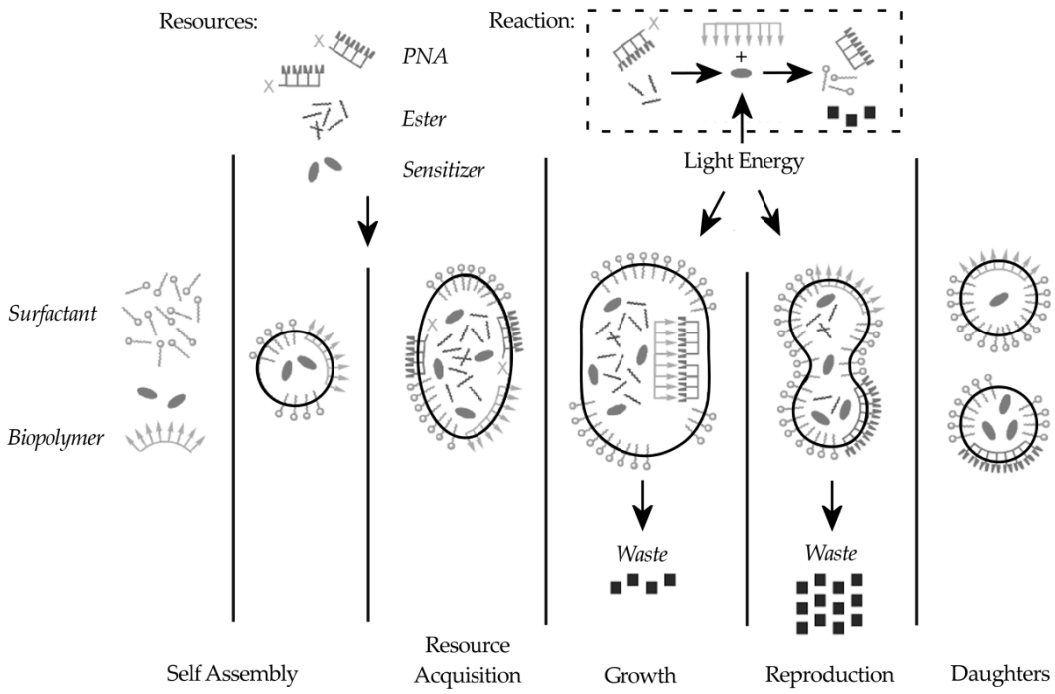



*Figure 2.3 Life cycle of the Los Alamos Bug. The concept uses peptide nucleic acid (PNA) as the information carrier, which is an artificially synthesized polymer similar to deoxyribonucleic acid (DNA), the hereditary molecule of existing living organisms. Redrawn based on a figure given in (Fellermann, Rasmussen, Ziock, & Solé, 2007).*

# 3 Coarse-Graining into DPD

For the study of a particular system by the DPD simulation technique, the functional form of the conservative force and either one of the dissipative and stochastic forces (the other will be fixed by the conditions given in Section 2.2) need to be determined, corresponding to the chosen coarse-graining level. This is a kind of "inverse problem", in that a DPD model has to be constructed from given system observables that are usually taken from atomistic simulations or real-world experiments, like radial distribution functions for the case of conservative interactions. In DPD simulations, this has often been done by heuristic selections and hand tuning, but techniques based on optimization algorithms minimizing a particular error function are also being devised; and there is a need for standard procedures for the derivation and adjustment of DPD models. This chapter provides an overview of existing major techniques for coarse-graining.

## 3.1 DPD coarse-graining techniques

As discussed in Section 2.4, the dissipative and stochastic interactions in the DPD model can be considered as a momentum-conserving thermostat for, and completely decoupled from, an arbitrary conservative potential.

Thus, the problem of finding the correct coarse-grained DPD model for the simulation of a system comprises two parts: the derivation of the conservative force (or equivalently, potential) and the derivation of the dissipative and stochastic forces, or the DPD thermostat. For fluids, the conservative potential is derived from known structural properties, most importantly the radial distribution function, whereas the dissipative and stochastic force components are often adjusted to match transport properties such as the diffusion coefficient.

## 3.2 Reconstruction of conservative potentials

In DPD simulations, one might use an explicitly defined analytical expression for the interaction potential describing the system to be studied, such as the Lennard-Jones potential (equation (1.5)) commonly used as an approximation for modeling noble gases, or other simple potentials used in abstract discussions or comparisons of various simulation techniques, sometimes even with no intention to approximate the behavior of a real-world substance. For cases where the simulation is intended to accurately replicate the properties of a specific substance (or substances, for multi-component simulations) and there are no theoretical models for the potential, the conservative potentials for use in DPD simulations are constructed so that they recreate the structural properties of the system that will be studied, usually measured in all-atom MD simulations.

Most techniques for the derivation of conservative interaction potentials use the radial distribution function (RDF) as the input, and are theoretically based on a result by Henderson (1974), stating that “for a given system under given conditions of temperature and density, two pair potentials which give rise to the same RDF cannot differ by more than an additive constant”. This assures that, once a potential reproducing the input RDF is obtained, it is unique; but does not guarantee that it will be found.

### 3.2.1 Inverse Monte Carlo

The inverse Monte Carlo (IMC) method (Lyubartsev & Laaksonen, 1995; Lyubartsev, Karttunen, Vattulainen, & Laaksonen, 2003) allows the reconstruction of conservative potentials from a given radial distribution function (RDF), through a systematic optimization procedure. In a coarse-graining procedure, this allows one to obtain the effective conservative potential that should be used in the coarse-grained model to yield the same structural properties (embodied by the RDF) measured in atomistic MD simulations or real-world experiments.

The RDF, also known as the pair correlation function or PCF, is an important observable that characterizes the local structure of fluids. It is based on the probability density of finding some particle situated at a distance $r$ from another particle,

$$\rho(r) = \langle \sum_i \delta(r - r_i) \rangle \,, \tag{3.1}$$

where $\delta(r)$ is the Dirac delta function and $i$ runs from 1 to $N$, the number of particles in the system. The RDF in a system of volume $V$ is given by normalizing this such that

$$g(r) = \frac{V}{N} \frac{\langle \sum_i \sum_{j \neq i} \delta(r - r_{ij}) \rangle}{4\pi r^2} = \frac{V}{4\pi r^2 N^2} \langle \sum_i \sum_{j \neq i} \delta(r - r_{ij}) \rangle \,, \tag{3.2}$$

i.e. the ratio of the average number density of particles at a distance $r$ from any given particle (within a spherical shell of volume $4\pi r^2$) to the density at a distance $r$ in an ideal gas at the same overall density ($N/V$). RDF is thus the conditional probability density of finding a particle at a distance $r$, given that there is a particle at the origin. By definition, $g(r) = 1$ for an ideal gas, and any deviation of $g(r)$ from unity reflects correlations between particles due to inter-particle interactions (Figure 3.1). The RDF plays an important role in theories of the liquid state. For instance, the average of any quantity that depends on pair distances $r_{ij}$ can be expressed as an integral over $g(r)$:

$$\langle A \rangle = \langle \sum_i \sum_{j>i} a(r_{ij}) \rangle = \frac{N^2}{2V} \int_0^\infty a(r) g(r) 4\pi r^2 dr \,. \tag{3.3}$$

As an example, the total average kinetic and potential energies of a monatomic system of particles interacting by a potential $\Phi(r)$ can be written as

$$E = \frac{3}{2} N k_B T + \frac{2\pi N^2}{V} \int_0^\infty \Phi(r) g(r) r^2 dr \,, \tag{3.4}$$

where the first term is the total kinetic energy by the equipartition theorem and the second term is the total potential energy integrated over $g(r)$.

The RDF is of particular interest because it is possible to measure it experimentally with neutron scattering or x-ray scattering experiments on simple fluids and light scattering experiments on colloidal suspensions. Coupled with techniques like the IMC described below, this allows for the inferring of effective interaction potentials from experimental observations.

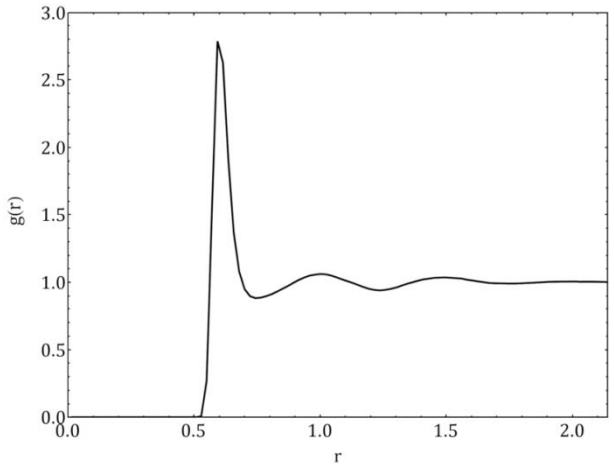



*Figure 3.1 An RDF plot for a liquid. A typical RDF plot has zero value at short separations up to a certain value specific to the system, representing the immediate "shell" around a molecule that is impenetrable because of strong repulsive forces. This is followed by a strong peak indicating the group of molecules packed around this shell. For a liquid, the RDF quickly converges to unity at longer distances, whereas the occurrence of sharp peaks at longer separations indicates a highly ordered system, such as a crystalline solid.*

The IMC method is basically a specialized version of the Newton-Raphson algorithm for finding roots of a system of equations, where at each iteration the error is computed as the difference of the target RDF from the RDF resulting from a Metropolis MC simulation (Section 1.2.2) of the system with the test potential. For a single-component system with pairwise interactions, the IMC method works as follows. The Hamiltonian for the system

$$H = \sum_{i,j} \Phi(r_{ij}) \,, \tag{3.5}$$

where $r_{ij}$ is the distance between particles $i$ and $j$ and $\Phi(r_{ij})$ is the pair potential, is discretized such that

$$H = \sum_{\alpha} \Phi_{\alpha} S_{\alpha} \,, \tag{3.6}$$

where $\alpha = 1, \dots, M$ is the discretization index with $M$ as the discretization resolution, $\Phi_{\alpha}$ is the constant step of the discretized potential between $r_{\alpha}$ and $r_{\alpha+1}$, and $S_{\alpha}$ is the number of particle pairs with a separation within the range $r_{\alpha}$ to $r_{\alpha+1}$, both with $r_{\alpha} = (\alpha - 1) r_c / M$ ($r_c$ is

the cut-off distance). In a similar fashion to equation (3.3), the average of $S_\alpha$ can be expressed in terms of $g(r)$,

$$\langle S_\alpha \rangle = 4\pi r_\alpha^2 \frac{r_c}{M} \frac{N^2}{2V} g(r_\alpha)\,, \tag{3.7}$$

for sufficiently large $M$. After the selection of the initial potential $\Phi_\alpha^{(0)}$, usually as $\Phi_\alpha^{(0)} = -k_B T \ln g(r_\alpha)$, the potential of mean force, the procedure works by repeating the following steps, starting with $t = 0$:

- Do a MC simulation with potential $\Phi_\alpha^{(t)}$ and compute the following
  - $\langle S_\alpha \rangle$, from the current RDF by equation (3.7)
  - $\Delta\langle S_\alpha \rangle = \langle S_\alpha \rangle - S_\alpha^*$ , with $S_\alpha^*$ given by the target RDF by equation (3.7)
  - The covariance matrix $\langle S_\alpha S_\gamma \rangle - \langle S_\alpha \rangle \langle S_\gamma \rangle$
- Compute error terms $\Delta\Phi_\gamma$ by solving the system of linear equations

$$\Delta\langle S_\alpha \rangle = \sum_\gamma \left( \frac{\partial \langle S_\alpha \rangle}{\partial \Phi_\gamma} \Delta\Phi_\gamma \right), \tag{3.8}$$

  using the relation

$$\frac{\partial \langle S_\alpha \rangle}{\partial \Phi_\gamma} = -\frac{\langle S_\alpha S_\gamma \rangle - \langle S_\alpha \rangle \langle S_\gamma \rangle}{k_B T} \tag{3.9}$$

- Unless the errors $\Delta\Phi_\alpha$ are sufficiently small (i.e. convergence is reached), update the potential $\Phi_\alpha^{(t+1)} = \Phi_\alpha^{(t)} - \lambda \Delta\Phi_\alpha$, increment $t$, and go to first step

The parameter $\lambda$ in the update step is a small number ($0 < \lambda < 1$) much like a "learning rate" in several optimization algorithms such as the backpropagation algorithm used in conjunction with neural networks. It facilitates the convergence of the procedure (Figure 3.2) by keeping the updates to the potential small in order to prevent overshooting the target potential causing oscillations in the iterations, particularly when the initial potential $\Phi_\alpha^{(0)}$ is differs greatly from the target potential.

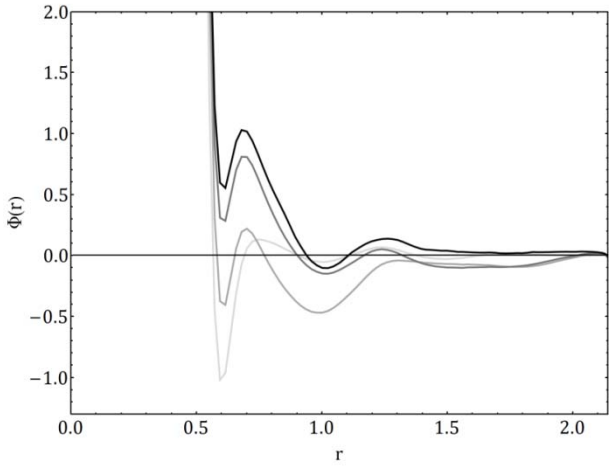



*Figure 3.2 The potential $\Phi(r)$ in the first three iterations of the IMC procedure with the sample RDF given in Figure 3.1, starting from potential of mean force $\Phi(r) = -k_B T \ln g(r)$ (lightest color). Iteration steps are represented with successively darker colors.*

In a DPD simulation, the potential $\Phi_\alpha$ that is converged upon by the IMC procedure is then numerically differentiated with respect to $r_\alpha$ (in a similar fashion to equation (1.3)) to get a discretized version of the conservative force component $F^C(r)$ (see Section 2.1).

### 3.2.2 Other techniques

Some other techniques for the determination of coarse grained potentials have been developed particularly for polymer simulations, such as the derivation of effective pair potentials from the RDF by using a hypernetted-chain (HNC) relation, by Louis, Bolhuis, Hansen, and Meijer (2000); and the work by Forrest and Suter (1995), in which rapidly fluctuating degrees of freedom in atomistic simulations are averaged over short time scales to obtain effective interaction potentials for coarse-grained MC simulations of polymers, a method they call "time coarse-graining".

In contrast to these systematic procedures, it has also been common in DPD literature to hand-tune the conservative interactions by making an arbitrary choice for the conservative force $F^C(r)$ so as to correspond to a simple repulsive soft potential, often in the form of equation (2.4). This function is then scaled (for the case of equation (2.4), the parameter $a$ is adjusted) to match the known compressibility of the system under study (Groot & Warren, 1997).

## 3.3 Adjustment of dissipative and stochastic forces

Just like the fact that the conservative interactions determine the structural properties of the system at equilibrium (like the RDF), picking the correct form for the dissipative and stochastic interactions is crucial for matching the known transport properties of the system studied by the DPD technique, such as diffusivity, thermal conductivity, shear viscosity and various autocorrelation functions.

### 3.3.1 Velocity autocorrelation

The velocity autocorrelation function (VAF) is an instance of time dependent correlation functions of the general form

$$C_a(t) = \langle a(0)a(t)\rangle\,, \tag{3.10}$$

which give the correlation between the values of a measured property at time 0, or the start of the measurement, and time $t$. The VAF is given by

$$C(t) = \langle \boldsymbol{v}_i(0)\cdot\boldsymbol{v}_i(t)\rangle\,, \tag{3.11}$$

where $\boldsymbol{v}_i$ is the velocity of a tracked particle $i$. A typical VAF plot for fluids (Figure 3.3) is calculated up to a certain value of $t = t_{max}$, around which the correlation between a particle's initial and current velocities decays to zero.

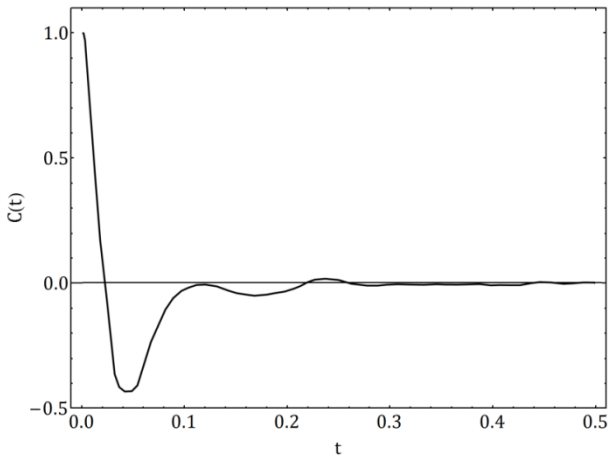



*Figure 3.3 A sample VAF plot, showing short-time bouncing and long-time decay.*

The VAF is an important observable revealing information about the underlying dynamics the particles in a simulation are subjected to and it has useful analytical properties. As examples, the Fourier transform of VAF can reveal underlying frequencies of molecular processes; and, through the Green-Kubo relations connecting correlation functions to transport coefficients (Green, 1954; Kubo, 1957), the diffusion coefficient $D$ in a system of dimensionality $d$ can be expressed as an integral over the VAF:

$$D = \frac{1}{d}\int_0^\infty \langle \boldsymbol{v}_i(0)\cdot\boldsymbol{v}_i(t)\rangle dt\,. \tag{3.12}$$

Lyubartsev, Karttunen, Vattulainen, and Laaksonen (2003) have used the VAF in their DPD coarse-graining study for adjusting the dissipative and stochastic forces, by trying to match the form of the VAF between DPD and atomistic MD simulations. They have done this by using a standard form for $\omega(r)$ and adjusting the dissipation parameter $\gamma$ in their system,

which was defined by equations (2.8) to (2.11), so that the short-time decay of the VAF is approximately the same in MD and DPD simulations (Figure 3.4).

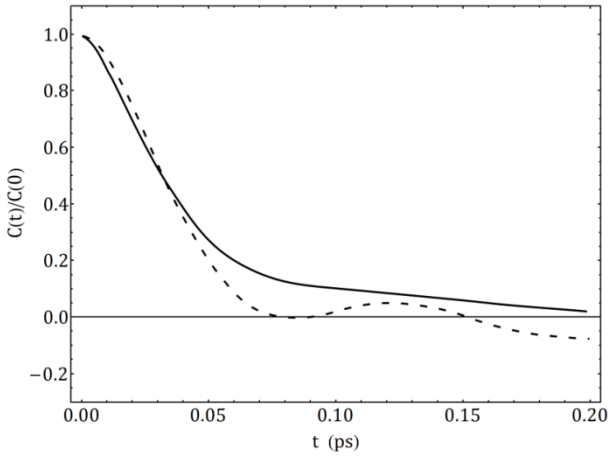



*Figure 3.4 Short time decay of the VAF for water, measured in an MD simulation (dashed curve) and a DPD simulation (solid curve) that is tuned to match the MD simulation. Reproduced based on (Lyubartsev, Karttunen, Vattulainen, & Laaksonen, 2003).*

### 3.3.2 Force covariance

Eriksson, Jacobi, Nyström, and Tunstrøm (2008b) suggested the use of the covariance of the forces (see equation (2.2)) acting on particles

$$\kappa_F(r) = -\Delta t \, \langle \boldsymbol{F}_i \cdot \boldsymbol{F}_j \rangle \big|_{r_{ij} = r} \tag{3.13}$$

as an observable for estimating $[\omega(r)]^2$, where $\Delta t$ is the time step used in the integration of the equations of motion (2.1) and $r_{ij}$ is the distance between particles $i$ and $j$. Measuring the force covariance $\kappa_F$ in DPD simulations with various forms of $\omega(r)$ and trying to recreate these by $\kappa_F$, they report that for time steps sufficiently small (smaller than usual for DPD simulations, close to $\Delta t = 10^{-3}$ in reduced units), but larger than the microscopic time scales that are represented as white noise after the DPD coarse-graining, $\kappa_F$ works as a remarkably good estimator for $[\omega(r)]^2$.

For cases where larger time steps are used, the measured $\kappa_F$ does not do well for approximating $[\omega(r)]^2$. In this situation, Eriksson et al. suggest making two measurements using different time steps $\Delta t_1$ and $\Delta t_2$, which lie within a region where $\kappa_F$ is an approximately linear function of $\Delta t$, and using a Richardson extrapolation for $[\omega(r)]^2$ at the limit of $\Delta t \to 0$:

$$[\omega(r)]^2|_{\Delta t = 0} \cong \frac{\Delta t_1 \Delta t_2}{\Delta t_2 - \Delta t_1} \Big[ \langle \boldsymbol{F}_i \cdot \boldsymbol{F}_j \rangle \big|_{\Delta t_2} - \langle \boldsymbol{F}_i \cdot \boldsymbol{F}_j \rangle \big|_{\Delta t_1} \Big], \tag{3.14}$$

which approximates the known exact form of $[\omega(r)]^2$ in their controlled experiment with an accuracy significantly better than either of the $\kappa_F$ measurements with $\Delta t_1$ and $\Delta t_2$ (Figure 3.5). It is also recommended that the linear region used in this extrapolation should not extend far beyond $\Delta t = 0.05$ in reduced units.

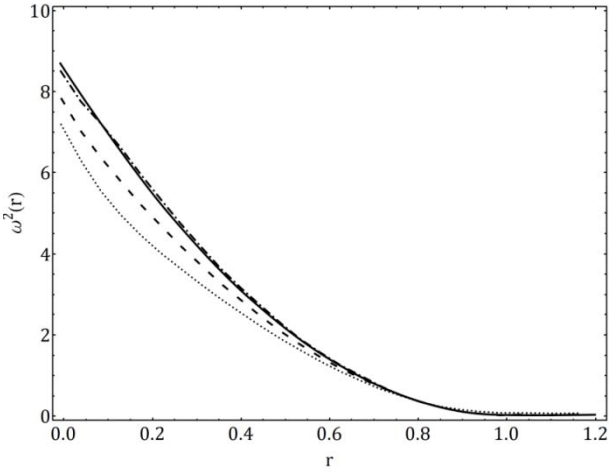



*Figure 3.5 Force covariance measurements in a DPD simulation, resulting from* $\Delta t = 0.025$ *(dashed curve) and* $\Delta t = 0.05$ *(dotted curve). Richardson extrapolation (dot-dashed curve) from these two measurements matches the exact form of* $[\omega(r)]^2 = [3(1-r)]^2$ *(solid curve) closely. Reproduced from (Eriksson, Jacobi, Nyström, & Tunstrøm, 2008b).*

### 3.3.3 Other techniques

The conventional method of adjusting the dissipative and stochastic forces has been to use a generic arbitrary form for $\omega(r)$—more than often the form in equation (2.13), justified as "a simple choice" (Hoogerbrugge & Koelman, 1992; Groot & Warren, 1997)—and then scaling it by trial and error to match some transport property that is deemed important for the particular type of study at hand, measured in a real-world experiment or an atomistic MD simulation.

Finding a generally applicable and proven procedure (like the IMC for the reconstruction of effective conservative potentials) for the determination of dissipative and stochastic part of dynamics in the DPD model is an active area of current research. In Chapter 4, we will examine the feasibility of using generic, black-box type optimization algorithms for the derivation of $F^C(r)$ and $\omega(r)$, using genetic algorithms as a case study.

# 4 DPD Coarse-Graining through Evolutionary Computation

This chapter starts with a brief introduction of evolutionary computation techniques, continuing with a description of the most commonly used member in this family, genetic algorithms. This optimization technique is used for the determination of coarse-grained interactions in the DPD model, based on fitness measures comparing equilibrium and transport properties of the system with those measured in atomistic simulations. As the model system, the *simple point charge* water model is used, for making the results comparable with selected literature. The technique is first used for the determination of conservative interactions from the RDF with the purpose of validating the approach by results from the IMC technique; and after that the dissipative interactions, based on escape time distributions. Through this case study, we will also gain information about the feasibility and performance of using generic optimization techniques in this domain.

## 4.1 Evolutionary computation

Evolutionary computation is a generic name given to the category of optimization algorithms based on Darwinian evolution, the process of gradual and hereditary changes of biological organisms over long periods of time, directed by natural selection in the environment which they live in. The algorithms grouped under this category emulate the process of biological evolution with different levels of fidelity, but all work by following the non-random survival of randomly changing solution candidates, based on a fitness (or alternatively, error) measure modeling natural selection. Evolutionary computation is also commonly identified as a subfield of artificial intelligence, as it allows machines to spontaneously devise solutions to problems, without human intervention.

Although there were computer simulations of evolution as early as 1954, the idea of applying evolutionary principles in computational models for problem solving matured in the following decades, most notably by the introduction of three independently developed techniques, namely, evolutionary programming (EP) (Fogel, 1964; Fogel, Owens, & Walsh, 1966), evolution strategies (ES) (Rechenberg, 1971; Schwefel, 1975), and genetic algorithms (GA) (Holland, 1975). The common feature of these techniques is that they work on a population of solution candidates that improve gradually with each passing generation, through a cycle of reproduction, modification, and selection.

The techniques of EP, ES, and GA, with many other variants thereof, are now classified as different members under the unified category of evolutionary algorithms (EA). The latest addition to the field is genetic programming (GP) (Koza, 1992), in which program trees—instead of simple data structures as in the case of GA—are subject to evolution, with genetic operators like mutation and crossover adapted to work on sub-branches of program trees and others like encapsulation and module acquisition added.

Within the general framework of optimization algorithms, the EA technique is classified as a type of metaheuristic combinatorial optimization that does not make any assumptions about

the error landscape underlying a particular problem—working as long as a fitness measure can be defined—and has thus been successfully applied in almost every subfield of engineering, natural sciences, and social sciences. The EA approach is particularly preferred because of its ability to provide fairly optimal solutions when there are no satisfactory problem-specific algorithms developed for a problem. These properties make it an interesting case study for the determination of interactions in the DPD model, with the possibility of using complex fitness functions measuring the equivalence of the coarse-grained model with the microscopic basis, incorporating equilibrium and time-dependent observables.

### 4.1.1 Genetic algorithms (GA)

A particularly common variety of evolutionary algorithms is genetic algorithms (GA) (Holland, 1975). As introduced in the previous section, descriptions of EA techniques employ basic concepts from evolutionary biology, such as population, fitness, selection, reproduction, and mutation. The flow of the standard GA, i.e. a version working on strings representing chromosomes and containing only the most basic genetic operations of mutation and simple crossover, proceeds as follows:

- Initialize the population, by creating a given number $n_{pop}$ of individuals (solution candidates) with randomly generated chromosomes (strings) of a given length $l_c$
- Evaluate the current population by assigning fitness values to all individuals using the problem-specific fitness measure, while keeping track of the highest fitness
- Form a new generation by doing the following until $n_{pop}$ new individuals are created
    - Based on the probability of crossover (sexual reproduction) $p_c$, do one of the following
        - Sexual reproduction: Select two individuals (with individuals with higher fitness having a higher probability of getting selected) and apply the crossover operator simulating sexual reproduction (Figure 4.1), producing two new individuals as offspring
        - Asexual reproduction (or cloning): Select an individual (with individuals with higher fitness having a higher probability of getting selected) and produce its copy as a new individual
    - Mutation: Apply the mutation operator (Figure 4.1) to the newly created individual (or individuals) with probability of mutation $p_m$
- Replace the population with newly created individuals, optionally keeping the individual with the highest fitness (a practice called *elitism*)
- Repeat from step two, until a given fitness threshold is reached by the individual with the highest fitness, or another end criterion is satisfied

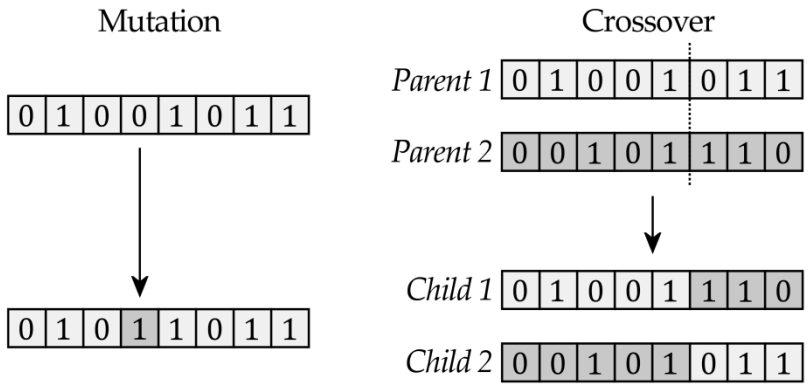



*Figure 4.1 Mutation and crossover genetic operators in standard GA.*

The GA workflow described above forms a generic framework that is applicable to a vast variety of optimization problems. The steps in preparing to apply it for the solution of a particular problem involve the definition of a problem-specific fitness measure and the decision of an encoding scheme through which the chromosomes (genotype) will be mapped to individuals to be evaluated (phenotype). The fitness measure is an algorithm used for assessing the performance of individuals in the environment, and can range from an evaluation of a simple mathematical function to a very long run of a highly complex simulation. The encoding scheme can either employ a direct one-to-one encoding of some parameters describing the individual, or a compact structure that will in turn create an individual by a simulated process of development, through techniques called artificial development or embryogenesis.

Another important consideration in designing a GA run is the selection procedure used for picking individuals from the population for reproduction. Two techniques are particularly common: The *roulette-wheel selection scheme* works by assigning each individual a slice of an imaginary selection wheel with a width directly proportional to its fitness value (the whole wheel representing the total of fitness of all individuals) and selecting a random angular position on the wheel. The *tournament selection scheme* involves, for each selection, the creation of a tournament group of size $n_t$ picked randomly from the population, arranging one-to-one contests in which the individual with the lower fitness is eliminated with probability $p_t$ (usually $> 0.75$), and designating the last standing individual as selected, loosely simulating real world population dynamics. The evolutionary parameters affecting the performance and convergence of the GA process, such as the population size, selection method, mutation rate $p_m$, and crossover rate $p_c$, are usually determined through simple heuristics and experience with former GA runs on similar classes of problems.

The most important advantages of GA—and EA in general—over classical optimization techniques like gradient descent, of which the IMC technique described in Section 3.2.1 for the determination of conservative potentials from RDFs is an instance, are that GA does not require the error function to be differentiable (see equations (3.8) and (3.9) of the IMC procedure) and that it is not bound to a deterministic trajectory over the error gradient and thus can escape local minima. In view of realization, the GA technique is particularly easy to implement and straightforward to parallelize.

## 4.2 Model experiment

In the following parts of this chapter, we will apply the GA technique for the derivation of coarse-grained conservative and stochastic interactions in a DPD simulation of *simple point charge (SPC) water*, with the aim of producing results comparable with the recent study by Eriksson, Jacobi, Nyström, and Tunstrøm (2008a), in which the transport properties in a united atoms (UA)[6] DPD simulation of SPC water were investigated.

The SPC water model, introduced by Berendsen, Postma, Van Gunsteren, and Hermans (1981), is a member in the large family (of more than 40) molecular models developed for simulations of the water molecule with varying degrees of detail and accuracy (Guillot, 2002). In the SPC model, water molecules ($H_2O$) are represented by three point charges corresponding to the one oxygen and two hydrogen atoms, interacting under Coulomb potential acting between all point charges and a Lennard-Jones potential (equation (1.5)) acting only between oxygen positions. The simplification of nuclei and electron orbits to point charges leads to an incorrect value for the dipole moment of the molecule, which is to some extent corrected by adjusting the H-O-H bond angle to 109.47° instead of its real value 104.45° known from experiments (Figure 4.2). The assumption of point charges also causes the SPC molecules to move somewhat faster than real water molecules (compare the diffusion rates in Table 4.1), but this effect decreases with increasing temperature. The model is rigid and the O-H bond lengths are exactly 1 Å.

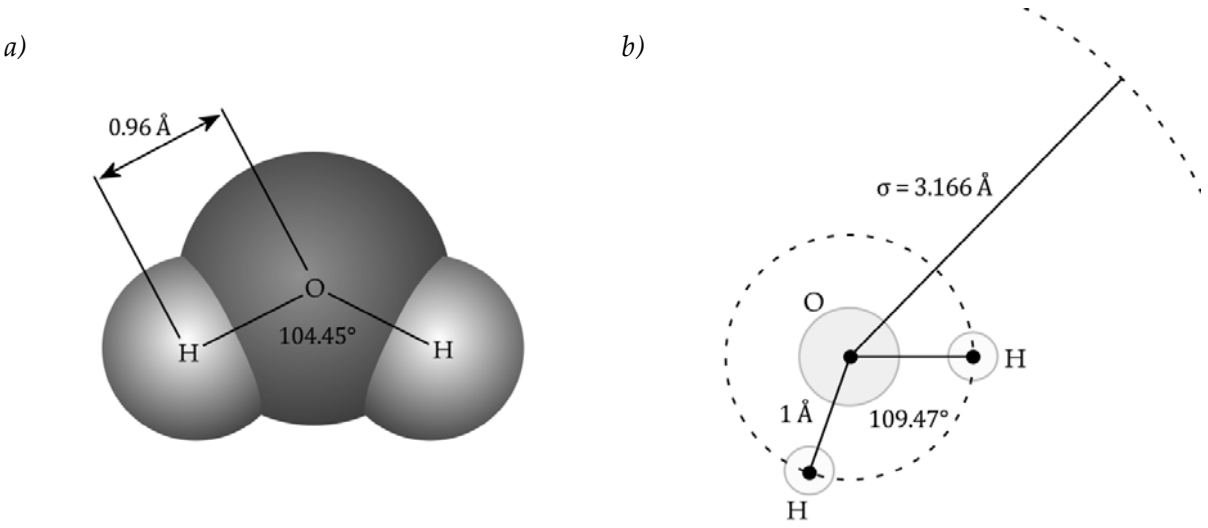



*Figure 4.2 a) Real water molecule. b) SPC water model, with Lennard-Jones parameters* $\sigma = 3.166\ Å$ *and* $\varepsilon = 0.650\ kJ\ mol^{-1}$.

A brief comparison of the properties of real water and SPC water are given in Table 4.1. The mass of one water molecule, $2.992 \times 10^{-26}$ kg, has been used as the base mass unit for reduced units (see Section 2.3) while performing DPD simulations. As in Eriksson et al., the

[6] The name "united atoms" denotes the practice of representing functional groups of bonded atoms within a molecule (or, for simple molecules like water, the whole molecule) with one pseudo-atom. This may be regarded as the first level of coarse-graining in super-atomic scale; on further levels, clusters of molecules may be represented by single pseudo-particles.

simulations have been performed at 298 K (24.85 °C), giving $k_B T \cong 1.381 \times 10^{-23} \times 298 = 4.115 \times 10^{-21}$ J, which was used as the base unit for energy. The length scale was defined in terms of $4.65 \times 10^{-10}$ m (4.65 Å), the approximate position of the potential well in the coarse-grained potential found by using the IMC method for UA SPC model (Figure 4.13).

*Table 4.1 Physical and thermodynamic properties of real and SPC water(Chaplin, 2008). All given values are at* 25 *°C (*298.15 *K) and* 1 *atm (*101.325 *kPa), except noted otherwise.*

| **Property** | **Real water** | **SPC water** |
|---|---|---|
| Dipole moment ($10^{-30}$ C m) | 9.84 (at 27 °C) | 7.57 |
| Relative static permittivity | 78.4 | 65 |
| Self diffusion ($10^{-5}$ cm $^2$ s $^{-1}$) | 2.30 | 3.85 |
| Average configurational energy (kJ mol $^{-1}$) | −41.5 | −41 |
| Density maximum (°C) | 3.984 | −45 |
| Expansion coefficient ($10^{-4}$ °C $^{-1}$) | 2.53 | 7.3 (at 27 °C) |

## 4.3 GA and DPD

The GA optimization technique is applicable for the determination of both the conservative and the dissipative part of interactions in a DPD simulation, while the ease of doing multiobjective optimizations with the GA technique, i.e. defining fitness functions involving more than one property of the coarse-grained system, allows one to incorporate both parts of dynamics in one optimization procedure, in contrast to existing techniques that were developed to apply specifically to either one of the two parts. Since there is the particularly successful and theoretically well-founded IMC technique for the construction of conservative potentials from the RDF, here we focus on the determination of the dissipative part of the dynamics and investigate the feasibility of having relatively long DPD simulations as part of fitness evaluations in a GA run and the success of the produced results compared to existing techniques.

Even if we focus on the dissipative part of the dynamics, we start by testing the GA approach on conservative interactions, providing an opportunity to test factors such as the selected encoding scheme, fitness evaluation algorithm, and evolutionary parameters, by validating the final result (the coarse-grained conservative potential) against the trusted one from the IMC technique, before going on to the determination of the dissipative interactions.

### 4.3.1 Representation of solutions

In order to apply the GA technique in this context, one of the first things that need to be defined is the template form for the functions to be optimized and the manner in which this will be encoded as a chromosome string.

A simple possibility is to use a parameterized simple function, such as equation (2.13) for $\omega(r)$, effectively handing over the parameter-tuning approach dominant in the DPD literature to the GA optimization. However, the versatility of the GA technique allows one to use much more complex and open-ended function templates; in fact, almost all models for curve fitting and regression analysis subfield of statistics can be employed for standing for solution candidates, including combinations of linear or nonlinear functions, polynomials, Fourier and power series, and splines. The main difference between curve fitting and this approach is that in curve fitting, the parameters of a chosen curve template are optimized so that an error measuring the proximity of the curve to a given set of data points is minimized, whereas in our study, there are no data points and the error is provided by the results of a DPD simulation using the curve ($\omega(r)$ or $F^C(r)$) as an input.

A very common choice in GA curve fitting studies is to use a $n$-degree polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \tag{4.1}$$

as the function template, and use a direct encoding for representing the coefficients $a_0, \ldots, a_n$ as real numbers in the chromosome. Through the use of variable length chromosomes, the decision of the required degree of polynomial can also be left to the GA optimization procedure. This approach is relatively successful in a curve fitting task, where a smooth low-degree polynomial (usually $n < 10$), not passing through all of the data points but having sufficient proximity to most of these, is often considered adequate. However, this technique suffers from a problem known as *Runge's phenomenon* (Figure 4.3 (a)), where violent oscillations in the function can appear, which can change the function being optimized radically even with very small changes in the GA genome. This was found to have adverse effects especially for the initial part of GA runs, i.e. the determination of conservative interactions from the RDF. A similar problem is also present in Fourier series, where it is called the *Gibbs phenomenon* (Figure 4.3 (b)).

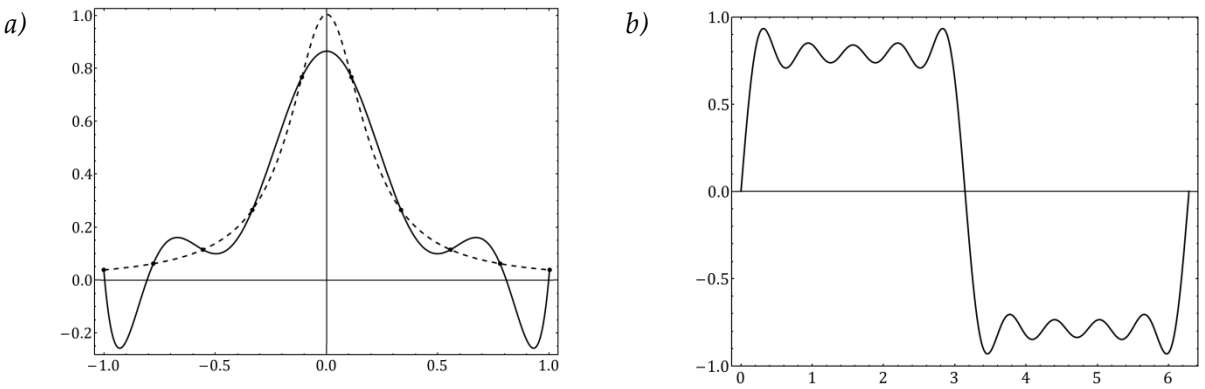



*Figure 4.3 a) Runge's phenomenon in a curve fitting example. A 10th degree polynomial (solid curve) is fitted to the 10 sample points on the function* $\frac{1}{1+25x^2}$ *(dashed curve). b) Gibbs phenomenon. The plot of the 5th degree Fourier series* $\sum_{i=1}^{5} \frac{1}{2i-1} sin[(2i-1)x]$ *approximating a square wave.*

In light of these considerations and GA test runs with other alternatives, it was decided to use a simple one-dimensional random walk as the function template. In contrast with a classical random walk taking successive random steps (relative to the previous step) that are accumulated as the walk progresses, we simply use a sequence of random numbers $\{x_\alpha\}$ representing the values of the function being optimized ($F^C(r)$ or $\omega(r)$) at the set of discrete $r$ values $\{r_\alpha\}$ (compare with equation (3.6)). This has the advantage of providing the optimization process with the ability of making local changes independent from the rest of the curve (via mutations) and merging parts of two existing curves into one simply by concatenating the parts together (via crossovers).

An intrinsic property of this simple random walk model is that it generates non-smooth functions consisting of piecewise linear sectors. However, the smoothness of $F^C(r)$ and $\omega(r)$ is especially important for the stability of the DPD algorithm with large time steps. One possibility for imposing the smoothness of the produced random walks is to include a measure of smoothness in the fitness measure, which would cause the GA procedure to eliminate random walks with highly non-smooth forms. Another one, as used in this study, is to apply some form of smoothing to the sequence $\{x_\alpha\}$. For obtaining smooth functional forms while still allowing relatively detailed local structure, we use Bézier splines[7] (Figure 4.4) fitted to the random walk, using the first and last points of the random walk as spline endpoints and the points in-between as control points. We call this approach for modeling arbitrary functions as *spline random walk* (Figure 4.5).

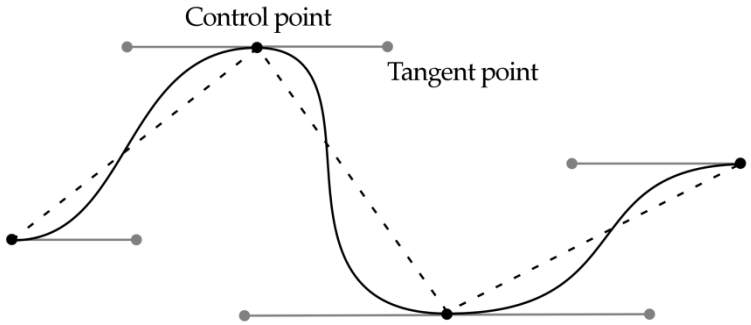



*Figure 4.4 A Bézier spline.*

The sequences $\{x_\alpha\}$ were encoded as real numbered chromosomes and in the first generation of the GA runs, initialized with uniform random numbers within a set range $(x_{min}, x_{max})$. A one-point crossover operator as shown in Figure 4.1 was employed. The standard mutation operator would change each position in a chromosome, with a given probability $p_m$, with a new random number in the range $(x_{min}, x_{max})$, causing abrupt changes in the form of the optimized function. After a number of trials, a special mutation operator was found to succeed considerably better, which scales the existing value at each mutated position by multiplication with a random number in the range $(1/2, 2)$. This allows mutations to produce gradual changes in sections of the function, facilitating the improvement of already good solutions.

[7] A Bézier spline is essentially a piecewise polynomial curve, whose shape is controlled by a given set of points. It is named after the French engineer Pierre Bézier, who publicized the method in 1960s for automobile design.

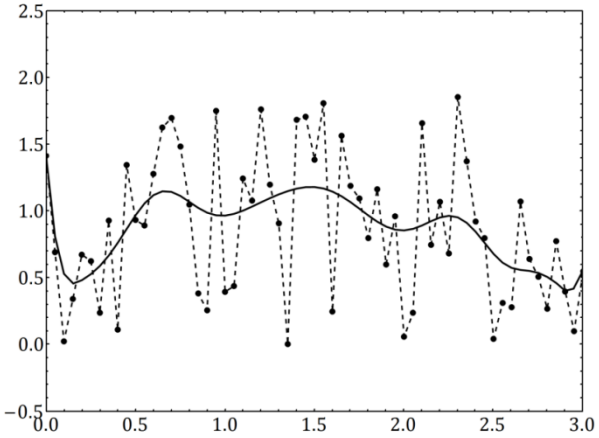



*Figure 4.5 Spline random walk with a Bézier spline (solid curve) fitted to a random walk of* 60 *points (points connected by dashed lines).*

### 4.3.2 Choice of fitness measures

For the first part of the GA runs determining the conservative interactions, i.e. optimizing $F^C(r)$, the difference between a given target RDF measured in atomistic SPC water simulation and the RDF produced at the end of a DPD simulation with the evaluated form of $F^C(r)$ has been used as the error measure, with lower errors translating to higher fitness values in the GA procedure. For the calculation of the error, the $L^2$-norm (also known as the Euclidean norm) has been used, which is given by

$$\sqrt{\sum_{\alpha=1}^{M}\left[g(r_\alpha) - g_{target}(r_\alpha)\right]^2}\,, \tag{4.2}$$

where $g(r)$ is the RDF resulting from the evaluated $F^C(r)$, $g_{target}(r)$ is the target RDF, $r_\alpha$ is the discretized distance, with $\alpha$ as the discretization index and $M$ as the discretization resolution (similar to the IMC procedure, equation (3.6)). The DPD simulations for the evolution of $F^C(r)$ were performed with the dissipative and stochastic interactions determined by a generic $\omega(r)$ supplied beforehand, bearing in mind that the form of the thermostat will not have an effect on the RDF values averaged at equilibrium.

For the second part, where the dissipative and stochastic interactions are determined through optimizing $\omega(r)$, the *escape time distribution* (ETD) was adopted as the observable, illustrating the probability of two particles getting separated by more than a given distance, depending on the passed time and their initial separation (Figure 4.6). Thus, an ETD plot gives a very detailed view of local dynamics, compared with simple scalar measures of time dependent properties, such as the diffusion coefficient and viscosity. Again, the Euclidean norm was used for computing the error between the ETD measured during fitness evaluations and the target ETD measured in the atomistic SPC water simulation. With a given discretization resolution $M$, the two-dimensional ETD plot was treated as a probability matrix and the error was defined as a $L^2$-norm running over all the elements of the two matrices:

$$\sqrt{\sum_{i=1}^{M}\sum_{j=1}^{M}\left[\underline{\boldsymbol{E}}_{i,j}-\underline{\boldsymbol{E}}_{target_{i,j}}\right]^2}\ . \tag{4.3}$$

For this part, in DPD simulations for the evaluation of fitness values by measuring the ETD resulting from a tested $\omega(r)$, the conservative force $F^C(r)$ found by the GA procedure in the previous part, or by the IMC procedure, has been used to set the conservative part of dynamics.

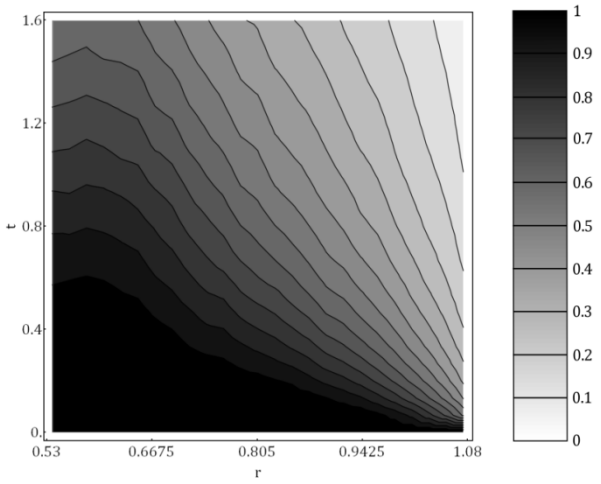



*Figure 4.6 Escape time distribution in a DPD simulation with* 2744 *particles. The plot gives the probability of two particles remaining within a distance of* 1.075, *given their initial separation* $r$ *and the passed time* $t$.

### 4.3.3 Fitness evaluations

During the GA runs, the fitness evaluation of a given chromosome (step two within the general GA procedure described in Section 4.1.1) consisted of the following main steps:

- Decode the chromosome, getting the random walk sequence $\{x_\alpha\}$ smoothed with a Bezier spline fit as described in Section 4.3.1
- Set up a DPD simulation
    - For evaluations of conservative force: Use $\{x_\alpha\}$ as $F^C(r)$, with a given $\omega(r)$
    - For evaluations of dissipative and stochastic forces: Use $\{x_\alpha\}$ as $\omega(r)$, with a given $F^C(r)$
- Equilibrate the DPD simulation for a given number of initialization time steps
- Take the measurements
    - For evaluations of conservative force: Sample RDF
    - For evaluations of dissipative and stochastic forces: Sample ETD
- Compute the error between measured and target observables, through equations (4.2) and (4.3)

For obtaining speedups in the optimization procedure, a fail condition for every fitness evaluation was introduced, checking the stability of the DPD algorithm with the tested functional form. This was achieved by checking the value of $k_B T$ (which should be close to 1 in reduced units, see Section 2.3) after the end of equilibration of DPD simulations, through the equipartition theorem

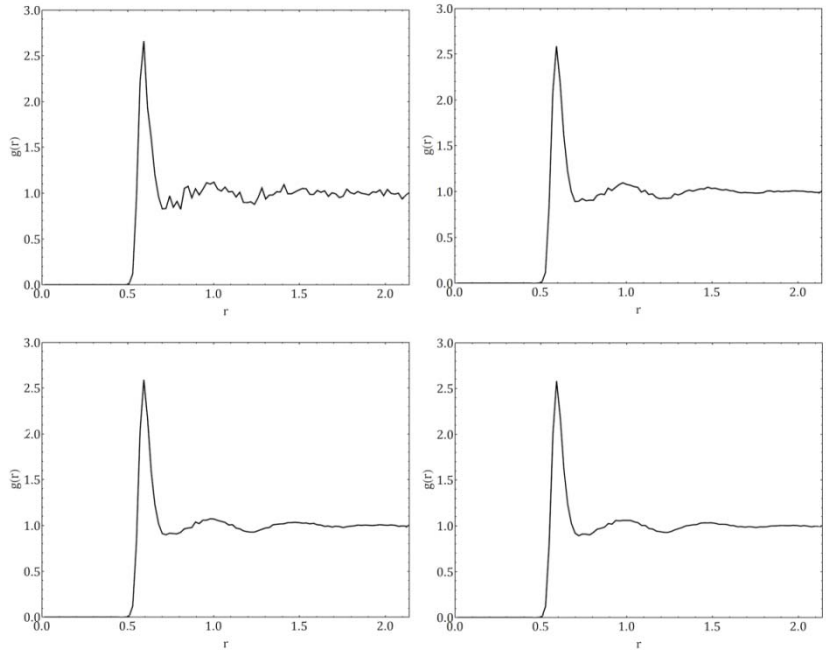
$$\sum_{i=1}^{N} m_i v_i^2 = 3N k_B T \,, \tag{4.4}$$

where $N$ is the number of particles, $m_i$ is the mass (equal to 1 in a one-component system with reduced units) and $v_i$ the speed of particle $i$. For individuals with $k_B T$ differing significantly from unity, the fitness evaluations were preemptively ended with a predefined worst fitness value. This was especially useful during the first few generations, where the tested functional forms are essentially random and do not necessarily conform to the stability requirements of the DPD procedure. With the help of this stability condition and the fitness measure assessing the physical plausibility of the system with RDF and ETD comparisons, virtually all individuals in later generations represent functional forms that are theoretically acceptable as the final result.

Because of the limited system size (i.e. number of particles) used in the simulations, single instances of RDF and ETD measurements are subject to noise. This was overcome by using the average of several samples as the final measurement. The total number of averaging steps for each observable were determined based on experiments with a typical DPD simulation with 2744 particles, observing the convergence of the averaged RDF and ETD into a smooth form, and also keeping in mind performance considerations (i.e. a longer than necessary number of averaging steps incurs a significant burden on the time needed for one GA evaluation). For the measurement of RDF 45 samples, and for ETD, 12 samples were used (Figure 4.7).

It should also be noted that the number of samples needed for averaging out the noise to a sufficient degree decreases in simulations with higher particle numbers. However, using higher number of particles demands more computation time per DPD simulation step, and this can easily surpass any gains by using less sampling steps. The optimum number of particles and the number of averaging steps should thus be determined by striking a balance between these two considerations.

*a)*




*b)*

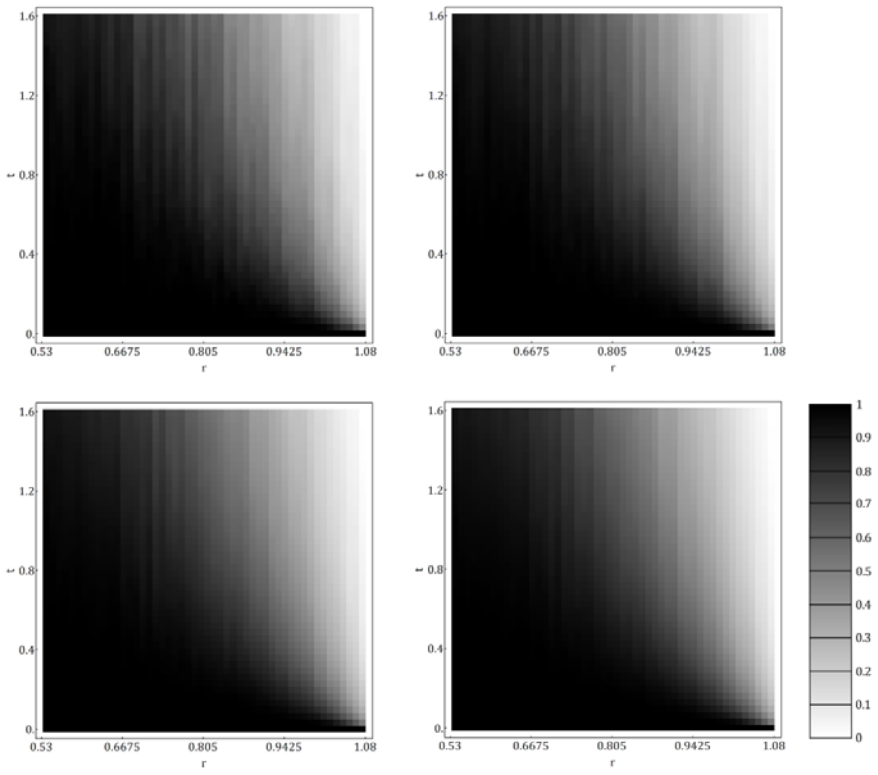



*Figure 4.7 a) The RDF averaged in a DPD simulation with* 1000 *particles. Averages of* 5, 15, 30, *and* 45 *samples, with* 50 *time step intervals between each sample. b) ETD averaged in the same simulation, with* 2, 4, 8, *and* 12 *samples, shown as a density plot to better illustrate the effect of averaging.*

All DPD simulations for fitness evaluations were started with an initial configuration of particles in a simple cubic lattice. The simulations were performed in cubic boxes and the

number of particles were selected to be perfect cubes (e.g. $1, 8, 27, \ldots, 2744, 3375, 4096, \ldots, n^3$), for having the same inter-particle distance in each Cartesian direction. As periodic boundary conditions (described in Section 1.2.1) were used, the minimum and maximum coordinates of the simulation box on each axis correspond to the same coordinates, thus, the initial lattice within the box is scaled so that particle overlaps are avoided and the system represents a perfect infinite cubic lattice filling all space (Figure 4.8).

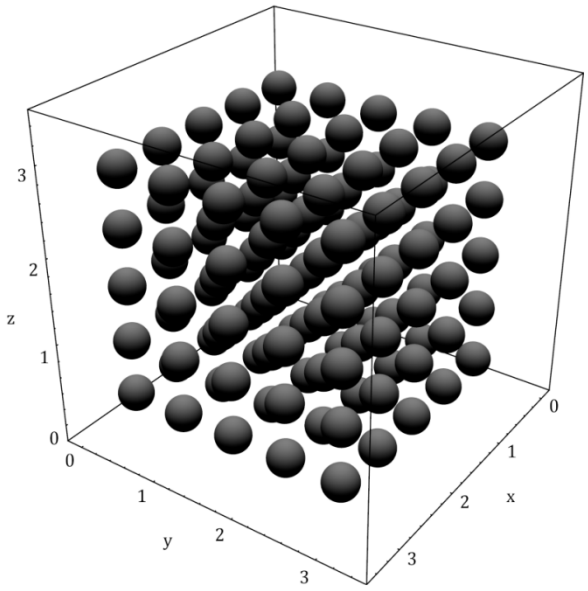



*Figure 4.8 Initial configuration with* 125 *particles, box side length* 3.368*, number density* 3.271*.*

## 4.4 Implementation

The core simulation components were coded in C# programming language (version 3.0), with the compiled assemblies running on Microsoft .NET Framework 3.5 on Microsoft Windows operating system. Some mathematical operations needed by the code, such as the solution of a system of linear equations for the IMC procedure (Section 3.2.1) and the computation of Bezier splines (Section 4.3.1), along with some parts of the user interface (Figure 4.9) such as function plots, were coded in the high-level *Mathematica* programming language. Communication between these two code bases were established using the .NET/Link protocol provided with the *Mathematica* 6.0 kernel by Wolfram Research.

In addition to the GA implementation (Figure 4.9) working as a standalone program accessing the *Mathematica* kernel for some operations, special *Mathematica* interfaces for the C# engine for setting up and running DPD simulations (Figure 4.10) and IMC optimizations (Figure 4.11) were also developed, allowing observation of the progress on the fly as the run proceeds.

The simulation components were implemented in a modular fashion, allowing different combinations of the subparts and future extensions with ease. The time integration engine was implemented as an abstract class employing the velocity Verlet algorithm (see equations (1.7)), which was then inherited by classes implementing MD and DPD models. A separate class was written for the Metropolis MC algorithm. The C# code uses three-dimensional

vector structures allowing the coding of vector operations very similar to how these appear in equations. The Verlet neighbor list optimization (Section 1.2.1) was used in a similar fashion to that described in (Chialvo & Debenedetti, 1992) and in code this has been implemented as an $N \times N$ adjacency matrix with binary entries.

The computer experiments were conducted on a 64-bit hardware platform (x86-64) with AMD Athlon 64 3200+ processor and 1.25 GiB random access memory. On this platform, a DPD run with 2744 particles for 1000 simulation time steps took approximately 130 seconds.

The source code of implementations are given in Appendix B.

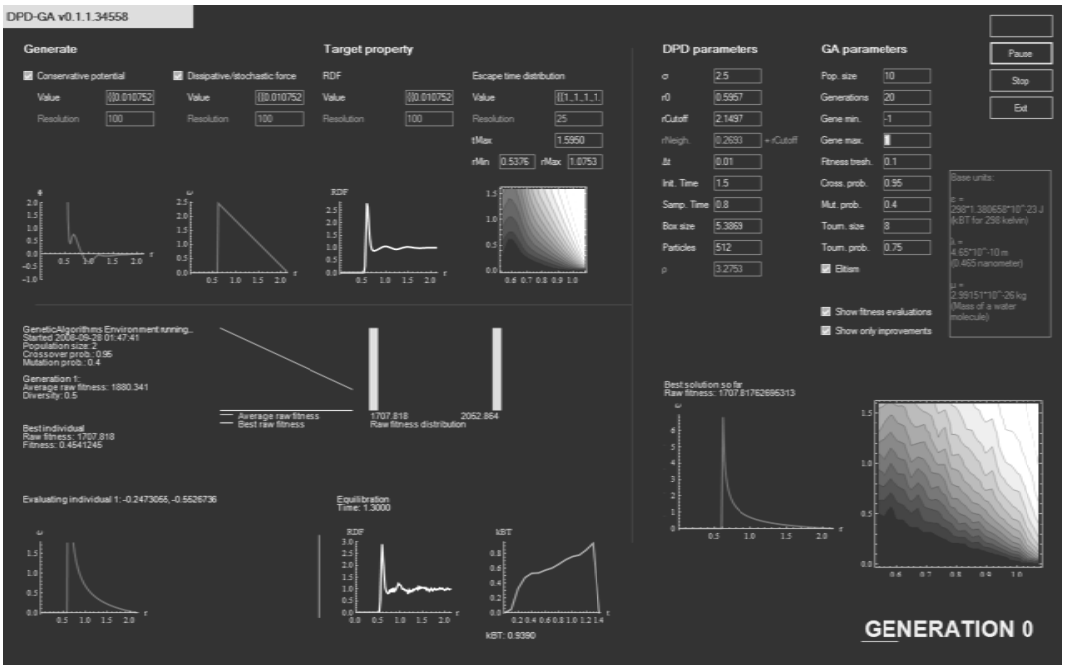



*Figure 4.9 Screenshot of the GA implementation evolving DPD interactions.*

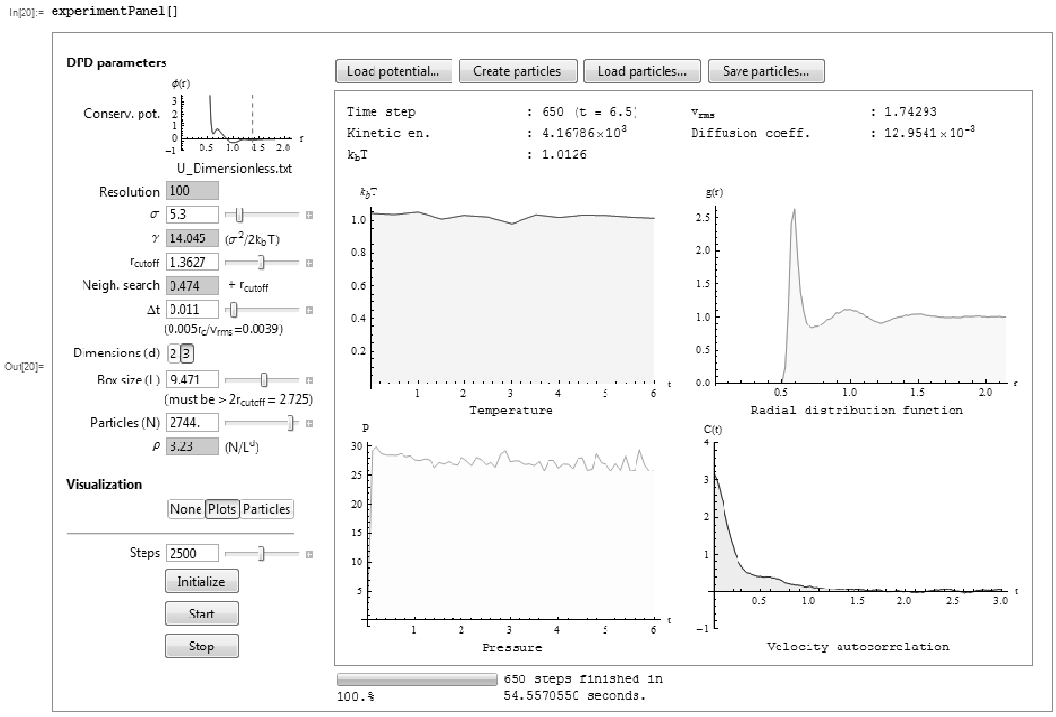



*Figure 4.10 The DPD simulation interface coded in Mathematica.*

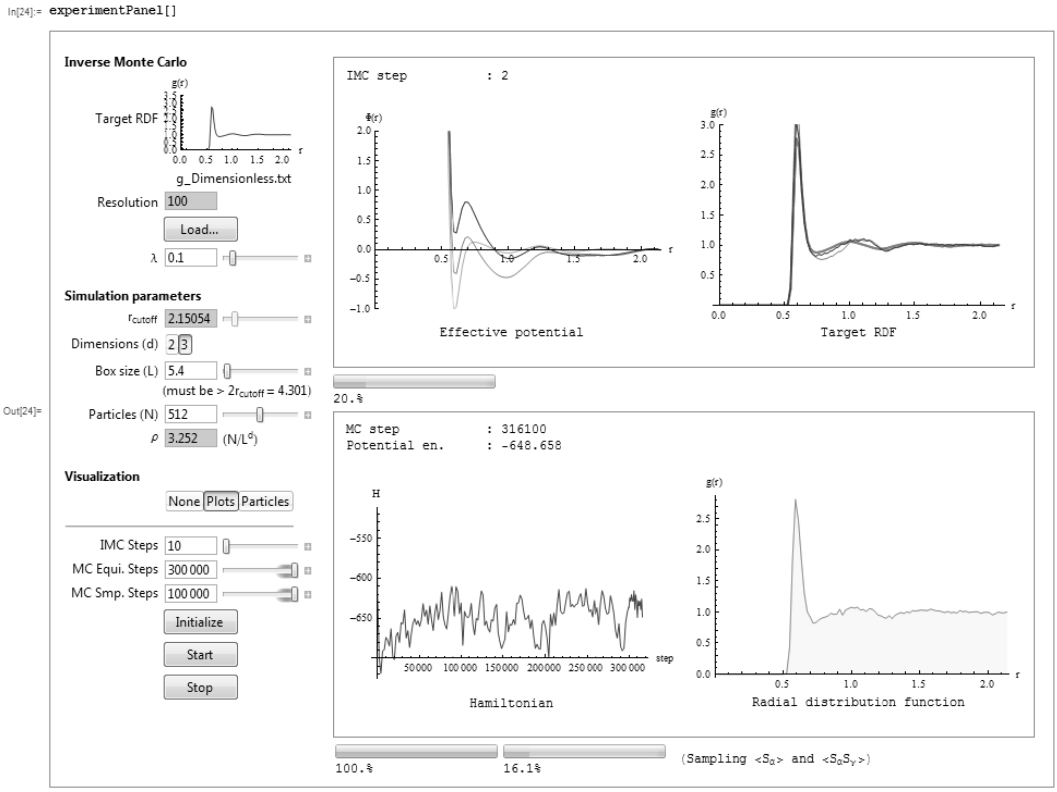



*Figure 4.11 The IMC interface coded in Mathematica.*

## 4.5 Results

### 4.5.1 Simulation setup

The parameters of the DPD simulations within GA evaluations are selected as to make the final results comparable with those by Eriksson, Jacobi, Nyström, and Tunstrøm (2008a). The simulations were performed, as described in Section 4.2, in reduced units. The simulation box length has been selected so that a number density of 3.275 (corresponding to 974.472 kg m $^{-3}$ in physical units) is achieved with 2744 particles, to match the density of simulations by Eriksson et al. This value of is close to the density of real water at 14 °C (287.15 K) under 1 atm pressure. Table 4.2 gives a list of DPD parameters that were common in all performed simulations.

*Table 4.2 DPD parameters common in simulations. Properties in parentheses are derived from others.*

| **Property** | **Value (reduced units)** | **Value (physical units)** |
|---|---|---|
| $N$, number of particles | 2744 | |
| $L$, simulation box side length | 9.427 | 4.384 nm |
| ($\rho_N$, number density) | 3.275 | 32.57 nm $^{-3}$ |
| ($\rho$, density) | | 974.472 kg m $^{-3}$ |
| $T$, Temperature | 1 | 298 K |
| $\Delta t$, time step | 0.01 | 0.0125 ps |
| Equilibration time | 3.5 | 4.388 ps |
| $r_0$, minimum distance | 0.596 | 0.277 nm |
| $r_c$, cutoff distance | 1.72 | 0.8 nm |
| $r_n$, neighbor search | 2.191 | 1.019 nm |

The GA parameters used in simulations are summarized in Table 4.3. Note that the gene minimum and maximum values only determine the lower and upper bounds of the random functions created to form the first generation, and functions can attain values outside this range through the scaling mutation procedure described in Section 4.3.1. The genome length $l_c$ determines the number of points in the spline random walks (i.e. tested functions) and also the sampling resolution of DPD measurements like the RDF and ETD (in both dimensions).

*Table 4.3 The GA parameters used in simulations.*

| **Property** | **Value** |
|---|---|
| $n_{pop}$, population size | 250 |
| $l_c$, genome length | 100 |
| Gene minimum | 0 |
| Gene maximum | 6.5 |
| $p_c$, crossover probability | 0.91 |
| $p_m$, mutation probability | 0.1 |
| $n_t$, tournament size | 8 |
| $p_t$, tournament probability | 0.75 |
| Elitism | Employed |

## 4.5.2 Comparison with existing techniques

### 4.5.2.1 Equilibrium properties

We start testing the practicality of the GA approach by trying to replicate the radial distribution function (RDF) measured in a MD simulation with SPC water, as described in Section 4.2. This is actually performed as a test for the success of the chosen fitness function and solution template (i.e. spline random walk) by validation of the results against the already established IMC technique (Section 3.2.1).

The DPD model used in fitness evaluations was formed by equations (2.1) – (2.3) and (2.12), with the form of dissipative and stochastic interactions given by

$$\omega(r) = \begin{cases} 0 & \text{if } 0 < r \leq r_0 \\ \sigma\left(1 - \dfrac{r}{r_c}\right) & \text{if } r_0 < r \leq r_c \\ 0 & \text{if } r > r_c \end{cases}, \tag{4.5}$$

where $r_0$ is a minimum threshold below which the dissipative and stochastic forces are zero, $r_c$ is the cutoff distance, and $\sigma$ determines the strength of stochastic interactions as usual. The parameter $r_0$, introduced by Eriksson et al. (2008a), acts as a restraint preventing the stochastic force from pushing particles within the region where the expected pair potential is practically infinity (as in Figure 3.2), or equally, the target RDF is zero (Figure 4.12). The value of $\sigma$ was taken as 2.589 in reduced units, corresponding to $4 \times 10^{-7}$ kg $^{1/2}$ s $^{-1/2}$ in physical units. The GA evaluations were based on a fitness function measuring the $L^2$-norm of the difference of the target RDF measured with SPC water and the RDF produced by the

evaluated system (Section 4.3.2). The RDF measurements were performed up to a distance $r = 2.150$ (1 nm). The actual function optimized by the GA procedure was the pair potential $\Phi(r)$, from which the conservative pair force $F^C(r)$ was derived by numerical differentiation to be used in DPD simulations.

Figure 4.12 shows the comparison of the target RDF with the RDF resulting from a pair potential $\Phi(r)$ found as the best result in a GA run after 21 generations of evolution (with the parameters outlined in Section 4.5.1). In Figure 4.13, this pair potential and its comparison with the result from the IMC procedure are presented.

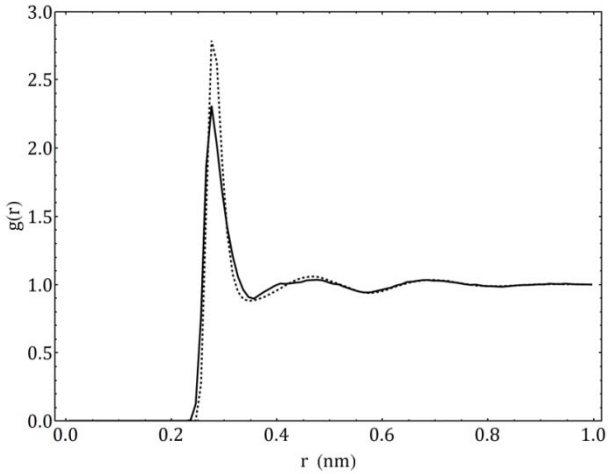



*Figure 4.12 The target RDF measured in MD simulation of SPC water (dashed curve) and the RDF measured in a DPD simulation with the conservative interactions given by a solution selected from the GA procedure (solid curve).*

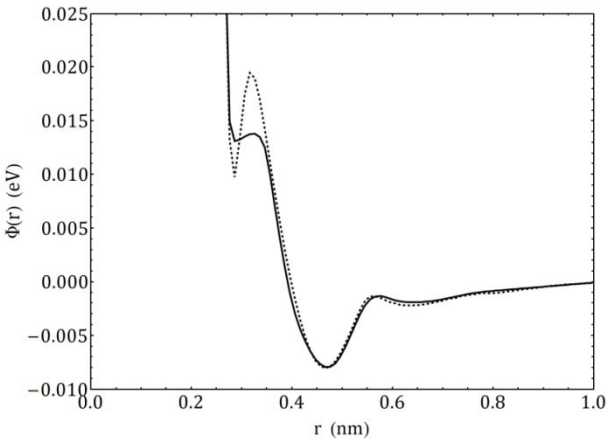



*Figure 4.13 Conservative pair potentials found by the GA procedure (solid curve) and the IMC procedure (dashed curve), using the RDF measured in MD simulation of SPC water.*

Sufficient agreement between the results from the GA and IMC procedures, and the fact that the results from the GA procedure would most probably keep improving in longer runs, suggest that the GA technique is applicable in this domain, with the chosen representation

and fitness measures. It is particularly of note that the technique was able to recreate most of the major features of the potential known from the IMC procedure, and it is highly likely that the form can be matched almost exactly in a longer GA run, since the form in Figure 4.13 is readily transformable to the one from the IMC procedure by a few local scaling mutations.

The most significant drawback of the method is the long computational time required to perform the fitness evaluations, lasting about 20 minutes per generation on the hardware configuration described in Section 4.4. This computational cost is clearly intolerable and unnecessary for the determination of conservative interactions from RDF, for which there is already the specialized IMC technique. Nevertheless, the cost is actually not prohibiting for using the technique to fill in parts of the adjustment process of DPD models, and other computational physics models in general, for which there currently are no standard procedures.

#### 4.5.2.2 Transport properties

The GA procedure was then applied for the determination of $\omega(r)$ controlling the form of dissipative and stochastic interactions in the DPD model, using the escape time distribution (ETD) measured in MD simulation of SPC water as the target property (Section 4.3.2). The ETD measurements were performed for particles with initial separations in the range $[0.537, 1.075]$ ([0.25 nm, 0.5 nm]) and for a time range of 1.595 (2 ps). The conservative interactions in DPD simulations were defined by the function $\Phi(r)$ determined by the IMC procedure (Figure 4.13).

The ETD produced by the best solution in a GA run with 26 generations is presented in Figure 4.14, together with the original measurement in SPC water and the one reported by Eriksson et al. (2008a) for their UA DPD simulation. Figure 4.15 gives the form of $\omega(r)$ defined by this selected solution by GA, and a comparison of this with the result reported by Eriksson et al. (producing the ETD in Figure 4.14 (b)), which was equation (4.5) with parameter $\sigma$ hand-tuned to match the diffusion rate and viscosity of the original SPC simulation.

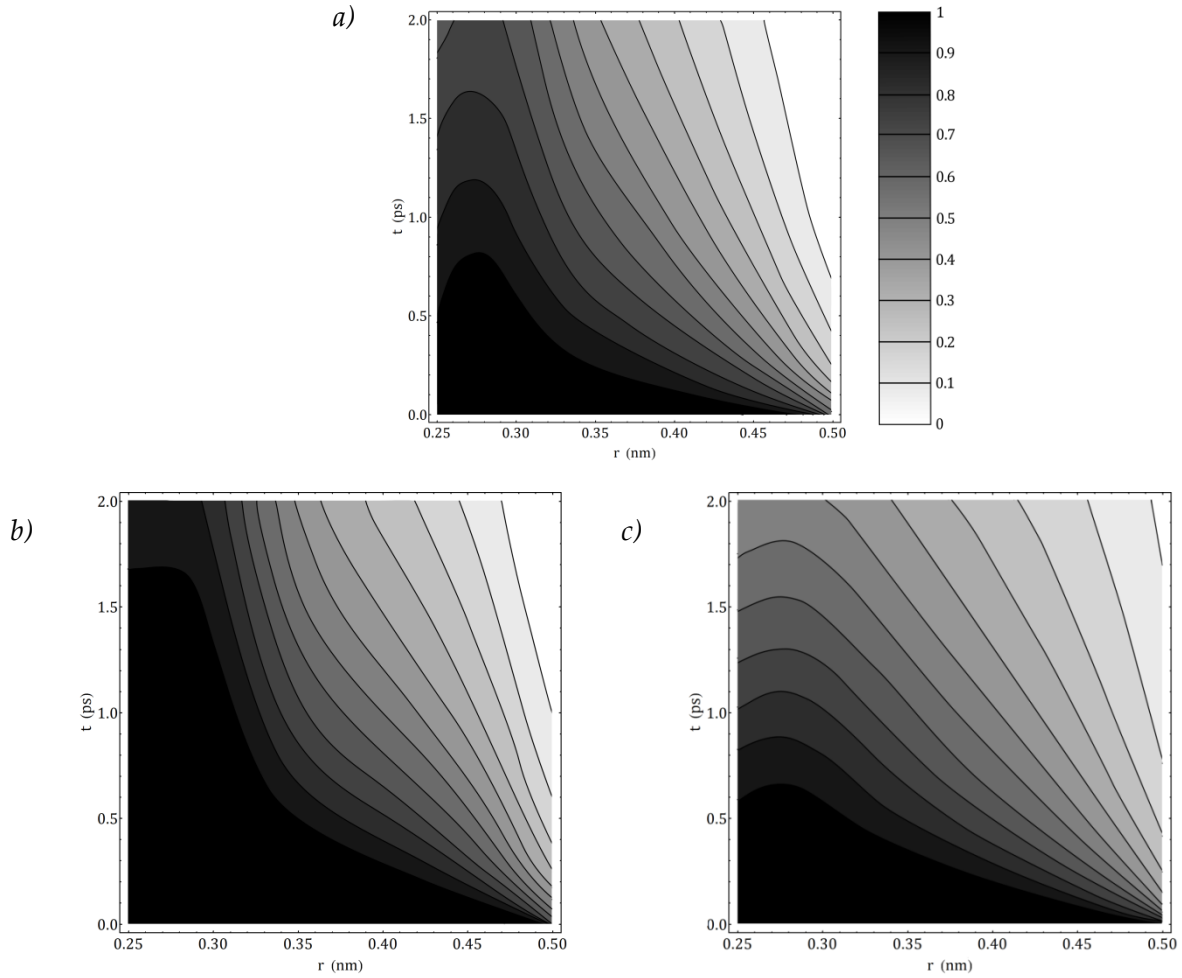



*Figure 4.14 ETD measurements for a) MD simulation of SPC water, b) UA DPD simulation of water by Eriksson et al. (2008a), and c) DPD simulation with the best $\omega(r)$ found by GA simulations.*

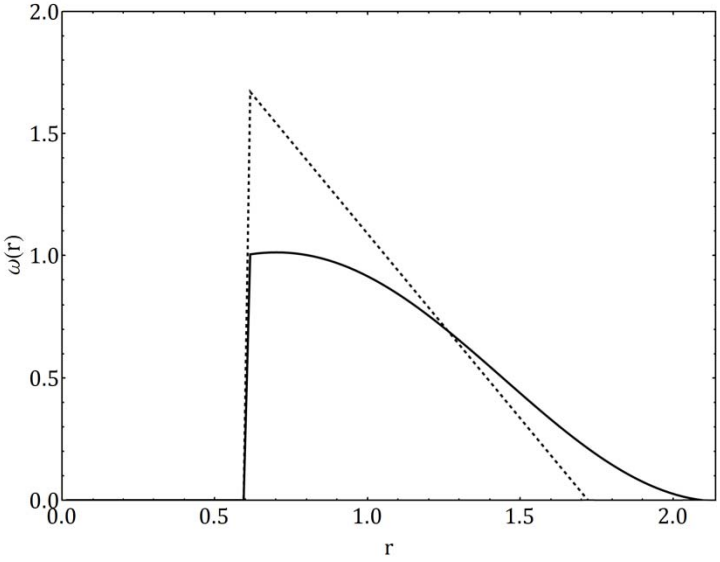



*Figure 4.15 Comparison of the form of* $\omega(r)$ *found by GA simulations (solid curve) with equation (4.5), used by Eriksson et al. (2008a), with* $\sigma = 2.589$, $r_0 = 0.596$, *and* $r_c = 1.72$ *in reduced units* ($\sigma = 4 \times 10^{-7}\ kg^{1/2}\ s^{-1/2}$, $r_0 = 0.277\ nm$, *and* $r_c = 0.80\ nm$) *(dashed curve).*

In approximating the reference MD simulation of SPC water, the ETD measurement resulting from the $\omega(r)$ found by the GA approach performs considerably better than using the generic linear form of $\omega(r)$ (equation (4.5)) with hand-tuned parameter $\sigma$. In Figure 4.14, it is particularly notable that the falling edge of the peak near $r = 0.25$ nm is reproduced by the GA approach, whereas this feature does not appear with equation (4.5). Unlike the common linear form of $\omega(r)$ normally used in DPD studies, the solution by the GA procedure has a non-linear form and behaves asymptotically at both ends of its domain. It is also notable that even if the used spline random walk template (Section 4.3.1) is capable of producing functions with detailed local structure (as with the conservative pair potential $\Phi(r)$ shown in Figure 4.13), and despite that such detailed candidates have been observed during the fitness evaluations, the final form of $\omega(r)$ eventually settled by the GA procedure has a very smooth form.

# 5 Conclusions

This study was a review of the DPD model and an investigation of the feasibility of using evolutionary optimization techniques for the determination of coarse-grained interactions from measurements in atomistic simulations. The review part included a very brief introduction to particle physics to set the stage for discussing the different levels on the coarse-graining ladder and the considerations that led to the development of the DPD model. After introducing basic methods in computational physics, such as MD and MC, the DPD technique was presented, including an account of its development and its application areas. We have also presented existing major techniques for coarse-graining into DPD.

Preliminary tests with SPC water showed that, for the determination of conservative interactions in DPD, GA simulations can replicate the results from the IMC technique to a sufficient degree, thus proving that the approach is credible; while the results for the dissipative and stochastic part of dynamics seem promising by performing better than the hand-tuning approach commonly used in literature. Overall, the performance of the GA technique suggests that it can be added into the toolset of coarse-graining techniques.

Besides confirming the feasibility of having relatively long DPD simulations within fitness evaluations of a GA procedure, this study also establishes a general framework for applying evolutionary optimization techniques for the determination of functional forms in possibly other models within the field of computational physics.

## 5.1 Further research

There are a number of directions that would be meaningful to explore as an extension to this study. To begin with, it would be interesting to see how the GA optimization procedure would perform with the addition of observables other than the RDF and ETD to the fitness evaluations, such as the velocity autocorrelation (described in Section 3.3.1) or the force covariance (described in Section 3.3.2). A particularly appealing experiment would be to investigate whether it is possible to combine the separately performed optimizations of the conservative and dissipative interactions into one complex fitness measure, including both RDF and ETD, and possibly other observables.

Again concerning the GA procedure, one could possibly devise better encoding schemes than the relatively ad hoc spline random walk; and it would also make sense to use parameterized analytical templates for $F^C(r)$ and $\omega(r)$ for specific applications, where such templates can be meaningfully defined from theory. Also, as is common with evolutionary optimization applications, domain-specific genetic operators could improve the convergence of the GA procedure, such as a crossover operator specialized for smoothly merging two partial functions.

Another study could be to investigate the performance of other evolutionary optimization techniques, most promisingly genetic programming (briefly mentioned in Section 4.1), which inherently produces functional expressions composed from a set of simple analytical functions. Obtaining function definitions for $F^C(r)$ or $\omega(r)$ would be much more useful as

generic results, in contrast with the discrete lists produced by the random walk GA and techniques like the IMC.

Lastly, due to the versatility of the evolutionary optimization approach and the ease with which fitness measures of arbitrary complexity can be formulated, it seems also promising to extend this work to many-body potentials, as the derivation of specific optimization techniques for these is analytically harder and most of the focus with existing techniques (such as the IMC) is on two-body potentials.

# References


Alder, B. J., & Wainwright, T. E. (1957). Phase transition for a hard sphere system. *Journal of Chemical Physics , 27*, 1208.

Anderson, E. C. (1999, October 20). *Lecture Notes for Stat 578C Statistical Genetics.* Retrieved September 6, 2008, from University of California, Berkeley; Department of Integrative Biology: http://ib.berkeley.edu/labs/slatkin/eriq/classes/guest_lect/mc_lecture_notes.pdf

Barbier, E. (1860). Note sur le problème de l'aiguille et le jeu du joint couvert (In French). *Journal de Mathématiques Pures et Appliquées , 2* (5), 273-286.

Berendsen, H. J., Postma, J. P., Van Gunsteren, W. F., & Hermans, J. (1981). Interaction models for water in relation to protein hydration. In B. Pullman (Ed.), *Intermolecular Forces* (pp. 331-342). Dordrecht: D. Reidel Publishing.

Boek, E. S., Coveney, P. V., & Lekkerkerker, H. N. (1996). Computer simulation of rheological phenomena in dense colloidal suspensions with dissipative particle dynamics. *Journal of Physics: Condensed Matter , 8*, 9509-9512.

Carlsson, A. E. (1990). Beyond pair potentials in transition metals and semiconductors. In H. Ehrenreich, & D. Turnbull (Eds.), *Solid State Physics: Advances in Research and Applications* (Vol. 43, pp. 1-91). New York: Academic.

Ceicedo-Carvajal, C. E., & Shinbrot, T. (2008). In silico zebrafish pattern formation. *Developmental Biology , 315* (2), 397-403.

Chaplin, M. (2008). *Water Models*. Retrieved 9 5, 2008, from Water Structure and Science: http://www.lsbu.ac.uk/water/models.html

Chen, S., Phan-Thien, N., Fan, X.-J., & Khoo, B. C. (2004). Dissipative particle dynamics simulation of polymer drops in a periodic shear flow. *Journal of Non-Newtonian Fluid Mechanics , 118* (1), 65-81.

Chialvo, A. A., & Debenedetti, P. G. (1992). An automated Verlet neighbor list algorithm with a multiple time-step approach for the simulation of large systems. *Computer Physics Communications , 70*, 467-477.

Dodd, A., & Dempsey, C. (2008). *Advanced Computing Research Centre Projects.* Retrieved September 2, 2008, from University of Bristol: http://www.acrc.bris.ac.uk/acrc/projects.htm

Dzwinel, W., & Yuen, D. A. (2000). Matching macroscopic properties of binary fluid to the interactions of dissipative particle dynamics. *Journal of Modern Physics C , 11* (1), 1-25.

Dzwinel, W., Boryczko, K., & Yuen, D. A. (2003). A discrete-particle model of blood dynamics in capillary vessels. *Journal of Colloid and Interface Science , 258* (1), 163-173.

Eriksson, A., Jacobi, M. N., Nyström, J., & Tunstrøm, K. (2008a). Effective thermostat induced by coarse graining of simple point charge water. *Journal of Chemical Physics , 129*, 024106.

Eriksson, A., Jacobi, M. N., Nyström, J., & Tunstrøm, K. (2008b). Using force covariance to derive effective stochastic interactions in dissipative particle dynamics. *Physical Review E, Statistical, Nonlinear, and Soft Matter Physics , 77*, 016707.

Español, P. (1998). Fluid particle model. *Physical Review E , 57* (3), 2930-2948.

Español, P. (1995). Hydrodynamics from dissipative particle dynamics. *Physical Review E , 52* (2), 1734-1742.

Español, P., & Warren, P. (1995). Statistical mechanics of dissipative particle dynamics. *Europhysics Letters , 30* (4), 191-196.

Fellermann, H., Rasmussen, S., Ziock, H.-J., & Solé, R. V. (2007). Life cycle of a minimal protocell: a dissipative particle dynamics study. *Artificial Life* , 319-345.

Fellermann, H., Rasmussen, S., Ziock, H.-J., & Solé, R. V. (2007). Life cycle of a minimal protocell: A dissipative particle dynamics study. *Artificial Life* , 319-345.

Filipovic, N., Kojic, M., & Tsuda, A. (2006). Modeling of microcirculation and thrombosis by Dissipative Particle Dynamics (DPD). *Journal of Biomechanics , 39* (1), S624.

Fogel, L. J. (1964). *On the Organization of Intellect (PhD Thesis).* Los Angeles: University of California, Los Angeles.

Fogel, L. J., Owens, A. J., & Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution.* John Wiley.

Forrest, B. M., & Suter, U. W. (1995). Accelerated equilibration of polymer melts by time-coarse-graining. *Journal of Chemical Physics , 102* (18), 7256.

Freddolino, P. L., Arkhipov, A. S., Larson, S. B., McPherson, A., & Schulten, K. (2006). Molecular dynamics simulations of the complete satellite tobacco mosaic virus. *Structure , 14*, 437-449.

Frisch, U., Hasslacher, B., & Pomeau, Y. (1986). Lattice-gas automata for the Navier-Stokes equation. *Physical Review Letters , 56* (14), 1505-1508.

Füchslin, R. M., Maeke, T., & McCaskill, J. S. (2007). Multipolar Reactive DPD: A Novel Tool for Spatially Resolved Systems Biology (Submitted).

Füchslin, R., Fellermann, H., Eriksson, A., & Ziock, H.-J. (2007). Coarse-graining and scaling in dissipative particle dynamics. *Journal of Chemical Physics (Submitted)* .

Gibson, J. B., Goland, A. N., Milgram, M., & Vineyard, G. H. (1960). Dynamics of radiation damage. *Physical Review , 120*, 1229.

Giordano, N. J., & Nakanishi, H. (2006). *Computational Physics.* Upper Saddle River: Pearson Prentice Hall.

Green, M. S. (1954). Markoff random processes and the statistical mechanics of time-dependent phenomena. II. Irreversible processes in fluids. *Journal of Chemical Physics , 22* (3), 398-413.

Groot, R. D. (2004). Applications of Dissipative Particle Dynamics. In M. Karttunen, I. Vattulainen, & A. Lukkarinen (Eds.), *Novel Methods in Soft Matter Simulations* (p. 2272). Springer-Verlag Berlin Heidelberg.

Groot, R. D., & Warren, P. B. (1997). Dissipative particle dynamics: Bridging the gap between atomistic and mesoscopic simulation. *Journal of Chemical Pysics , 107* (11), 4423-4435.

Guillot, B. (2002). A reappraisal of what we have learnt during three decades of computer simulations on water. *Journal of Molecular Liquids , 101* (1-3), 219-260.

Hadjiconstantinou, N. (2006). *22.00J/1.021J/2.030J/3.021J/10.333J/18.361J/HST.558J, Introduction to Modeling and Simulation, Spring 2006; Lecture notes 31: MD Simulation of Fluids.* Retrieved June 11, 2008, from Massachusetts Institute of Technology: MIT OpenCourseWare: http://ocw.mit.edu/OcwWeb/Nuclear-Engineering/22-00JSpring-2006/LectureNotes/index.htm

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika , 57* (1), 97-109.

Henderson, R. L. (1974). A uniqueness theorem for fluid pair correlation functions. *Physics Letters A , 49* (3), 197-198.

Heyes, D. M., Baxter, J., Tüzün, U., & Qin, R. S. (2004). Discrete-Element Method Simulations: From Micro to Macro Scales. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences , 362* (182), 1853-1865.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* Ann Arbor: University of Michigan Press.

Hoogerbrugge, P. J., & Koelman, J. M. (1992). Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics. *Europhysics Letters , 19* (3), 155-160.

Hoover, W. G. (1985). Canonical dynamics: Equilibrium phase-space distributions. *Physical Review A , 31* (3), 1695-1697.

Kim, J. M., & Phillips, R. J. (2004). Dissipative particle dynamics simulation of flow around spheres and cylinders at finite Reynolds numbers. *Chemical Engineering Science , 59* (20), 4155-4168.

Koelman, J. M., & Hoogerbrugge, P. J. (1993). Dynamics simulations of hard-sphere suspensions under steady shear. *Europhysics letters , 21* (3), 363-368.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Krauth, W. (2006). *Statistical Mechanics: Algorithms and Computations.* New York: Oxford University Press.

Kubo, R. (1957). Statistical-mechanical theory of irreversible processes. I. General theory and simple applications to magnetic and conduction problems. *Journal of the Physical Society of Japan , 12*, 570-586.

Louis, A. A., Bolhuis, P. G., Hansen, J. P., & Meijer, E. J. (2000). Can polymer coils be modeled as "soft colloids"? *Physical Review Letters , 85* (12), 2522-2525.

Lyubartsev, A. P., & Laaksonen, A. (1995). Calculation of effective interaction potentials from radial distribution functions: A reverse Monte Carlo approach. *Physical review E, Statistical physics, plasmas, fluids, and related interdisciplinary topics , 52*, 3730-3737.

Lyubartsev, A. P., Karttunen, M., Vattulainen, I., & Laaksonen, A. (2003). On coarse-graining by the inverse Monte Carlo method: Dissipative particle dynamics simulations made to a precise tool in soft matter modelling. *Soft Materials , 1* (1), 121-137.

Marx, D., & Hutter, J. (2000). Ab initio molecular dynamics: Theory and implementation. In J. Grotendorst (Ed.), *Modern Methods and Algorithms of Quantum Chemistry, NIC Series* (Vol. 1, pp. 301-449). Jülich: John von Neumann Institute for Computing.

Metropolis, N. (1987). The Beginning of the Monte Carlo Method. *Los Alamos Science* , 125-130.

Metropolis, N., & Ulam, S. (1949). The Monte Carlo method. *Journal of the American Statistical Association , 44*, 335-341.

Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics , 21*, 1087-1092.

Nakamura, H., & Tamura, Y. (2005). Phase diagram for self-assembly of amphiphilic molecule C12E6 by dissipative particle dynamics simulation. *Computer Physics Communications , 169* (1-3), 139-143.

Nosé, S. (1984). A unified formulation of the constant temperature molecular dynamics methods. *Journal of Chemical Physics , 81* (1), 511.

PACE Consortium. (2007). *Research Overview*. Retrieved June 5, 2008, from Programmable Artificial Cell Evolution (PACE): http://www.istpace.org

Protocell Assembly Project. (2004). *Protocell Assembly (PAs)*. Retrieved July 12, 2008, from Los Alamos National Laboratory: http://protocells.lanl.gov

Rahman, A. (1964). Correlations in the motion of atoms in liquid argon. *Physical Review , 136*, 405-411.

Rapaport, D. C. (2004). *The Art of Molecular Dynamics Simulation* (2nd ed.). Cambridge: Cambridge University Press.

Rasmussen, S., Bedau, M. A., Chen, L., Deamer, D., Krakauer, D. C., Packard, N. H., et al. (2008). *Protocells: Bridging Nonliving and Living Matter*. Cambridge: MIT Press.

Rechenberg, I. (1971). *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (PhD thesis, in German).* Berlin: Technical University of Berlin.

Schlijper, A. G., Hoogerbrugge, P. J., & Manke, C. W. (1995). Computer simulation of dilute polymer solutions with the dissipative particle dynamics method. *Journal of Rheology , 39* (3), 567-579.

Schneider, T., & Stoll, E. (1978). Molecular-dynamics study of a three-dimensional one-component model for distortive phase transitions. *Physical Review B , 17*, 1302-1322.

Schwefel, H.-P. (1975). *Evolutionsstrategie und numerische Optimierung (PhD thesis, in German).* Berlin: Technical University of Berlin.

Soddemann, T., Dünweg, B., & Kremer, K. (2003). Dissipative particle dynamics: A useful thermostat for equilibrium and nonequilibrium molecular dynamics simulations. *Physical Review E , 68*, 046702.

Succi, S. (2001). *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Oxford University Press.

Sutmann, G. (2002). Classical molecular dynamics. In J. Grotendorst, D. Marx, & A. Muramatsu (Eds.), *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes, NIC Series* (Vol. 10, pp. 211-254). Jülich: John von Neumann Institute for Computing.

Swope, W. C., Andersen, H. C., Berens, P. H., & Wilson, K. R. (1982). A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters. *Journal of Chemical Physics , 76*, 637.

Trofimov, S. Y. (2003). *Thermodynamic consistency in dissipative particle dynamics (PhD Thesis).* Technische Universiteit Eindhoven. Eindhoven: Eindhoven University Press.

Ulam, S., Richtmyer, R. D., & von Neumann, J. (1947). *Statistical methods in neutron diffusion.* Los Alamos Scientific Laboratory report LAMS-551.

Weinberg, S. (2003). *The Discovery of Subatomic Particles.* Cambridge: Cambridge University Press.

Veltman, M. (2003). *Facts and Mysteries in Elementary Particle Physics.* Singapore: World Scientific Publishing.

Verlet, L. (1967). Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Physical Review , 159*, 98-103.

Whittle, M., & Dickinson, E. (2001). On simulating colloids by dissipative particle dynamics: Issues and complications. *Journal of Colloid and Interface Science , 242* (1), 106-109.

Wu, H., Xu, J., Xianfeng, H., Yuehong, Z., & Wen, H. (2006). Mesoscopic simulation of self-assembly in surfactant oligomers by dissipative particle dynamics. *Colloids and Surfaces A: Physicochemical and Engineering Aspects , 290* (1-3), 239-246.

# Appendix A – Algorithms

## Derivation of the Verlet algorithm

For a system described by the equations of motion

$$\begin{aligned} \frac{\partial \boldsymbol{r}_i}{\partial t} &= \boldsymbol{v}_i \,, \\ \frac{\partial \boldsymbol{v}_i}{\partial t} &= \boldsymbol{a}_i \,, \end{aligned} \tag{B.1}$$

where $\boldsymbol{r}_i$, $\boldsymbol{v}_i$, and $\boldsymbol{a}_i$ are the position, velocity, and acceleration vectors of particle $i$; the common derivation of the Verlet algorithm (Verlet, 1967) for time integration starts with two third-order Taylor expansions of the positions $\boldsymbol{r}_i$:

$$\begin{aligned} \boldsymbol{r}_i(t+\Delta t) &= \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2 + \frac{1}{6}\boldsymbol{j}_i(t)\Delta t^3 + O(\Delta t^4) \,, \\ \boldsymbol{r}_i(t-\Delta t) &= \boldsymbol{r}_i(t) - \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2 - \frac{1}{6}\boldsymbol{j}_i(t)\Delta t^3 + O(\Delta t^4) \,, \end{aligned} \tag{B.2}$$

where $\boldsymbol{j}_i = \frac{\partial \boldsymbol{a}_i}{\partial t}$ is the jerk, or the third derivative of the position of particle $i$ with respect to time, and $\Delta t$ is the integration time step. Adding equations (B.2) yields

$$\boldsymbol{r}_i(t+\Delta t) = 2\boldsymbol{r}_i(t) - \boldsymbol{r}_i(t-\Delta t) + \boldsymbol{a}_i(t)\Delta t^2 + O(\Delta t^4) \,, \tag{B.3}$$

which is the basic form of the Verlet algorithm, giving the positions at the next time step $\boldsymbol{r}_i(t+\Delta t)$ based on the acceleration $\boldsymbol{a}_i(t)$ and the positions in the previous two time steps, $\boldsymbol{r}_i(t)$ and $\boldsymbol{r}_i(t-\Delta t)$, with an error of the order of $\Delta t^4$.

An issue with the basic Verlet algorithm is that the velocities $\boldsymbol{v}_i$ are not explicitly generated, while these are usually needed for measurements, such as for the calculation of the kinetic energy of the system. It is straightforward to compute these from the positions using the relation

$$\boldsymbol{v}_i(t) = \frac{\boldsymbol{r}_i(t+\Delta t) - \boldsymbol{r}_i(t-\Delta t)}{2\Delta t}, \tag{B.4}$$

with an error of the order of $\Delta t^2$.

## Derivation of the velocity Verlet algorithm

The standard Verlet algorithm described in the previous section has the undesirable property that for the computation of positions at time $t + \Delta t$, the positions at time $t$ and $t - \Delta t$ should be kept track of. The velocity Verlet algorithm (Swope, Andersen, Berens, & Wilson, 1982) is designed so that it alleviates this problem, i.e. it can start running from a given configuration

at just one time step, and provides both the positions $\boldsymbol{r}_i$ and velocities $\boldsymbol{v}_i$ explicitly at every time step.

For equations of motion (B.1), one can derive the velocity Verlet algorithm by starting with an expansion of $\boldsymbol{r}_i$:

$$\boldsymbol{r}_i(t+\Delta t) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2 + O(\Delta t^3) \tag{B.5}$$

and similarly for $\boldsymbol{v}_i$:

$$\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) + \boldsymbol{a}_i(t)\Delta t + \frac{1}{2}\boldsymbol{j}_i(t)\Delta t^2 + O(\Delta t^3)\,. \tag{B.6}$$

To write the jerk $\boldsymbol{j}_i = \frac{\partial \boldsymbol{a}_i}{\partial t}$ in equation (B.6) in terms of $\boldsymbol{a}_i$, one can use the relation

$$\boldsymbol{a}_i(t+\Delta t) = \boldsymbol{a}_i(t) + \boldsymbol{j}_i(t)\Delta t + O(\Delta t^2)\,, \tag{B.7}$$

which, when multiplied by $\frac{\Delta t}{2}$, gives

$$\frac{1}{2}\boldsymbol{j}_i(t)\Delta t^2 = \frac{1}{2}[\boldsymbol{a}_i(t+\Delta t) - \boldsymbol{a}_i(t)]\Delta t + O(\Delta t^3)\,. \tag{B.8}$$

Putting this into equation (B.6) yields

$$\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i(t) + \frac{1}{2}[\boldsymbol{a}_i(t+\Delta t) + \boldsymbol{a}_i(t)]\Delta t + O(\Delta t^3)\,, \tag{B.9}$$

which, with equation (B.5), forms the basic form of the velocity Verlet algorithm. The equations (B.5) and (B.9) are usually put into the form

$$\begin{aligned}
&\boldsymbol{r}_i(t+\Delta t) = \boldsymbol{r}_i(t) + \boldsymbol{v}_i(t)\Delta t + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t^2\,,\\
&\boldsymbol{v}_i\left(t+\frac{\Delta t}{2}\right) = \boldsymbol{v}_i(t) + \frac{1}{2}\boldsymbol{a}_i(t)\Delta t\,,\\
&\boldsymbol{a}_i(t+\Delta t) = \frac{\boldsymbol{F}_i(t+\Delta t)}{m_i}\,,\\
&\boldsymbol{v}_i(t+\Delta t) = \boldsymbol{v}_i\left(t+\frac{\Delta t}{2}\right) + \frac{1}{2}\boldsymbol{a}_i(t+\Delta t)\Delta t\,,
\end{aligned} \tag{B.10}$$

splitting the velocity updates into two parts by defining a half-step velocity $\boldsymbol{v}_i\left(t+\frac{\Delta t}{2}\right)$ and updating the accelerations $\boldsymbol{a}_i$ in-between, using forces $\boldsymbol{F}_i$. The algorithm, with the elimination of velocities $\boldsymbol{v}_i$, is mathematically equivalent to the standard Verlet algorithm, meaning that it computes exactly the same trajectories for the same initial conditions of a system.

# Appendix B – Source Codes

## MD and DPD

### Verlet engine

The code includes the abstract `Engine` class implementing the velocity Verlet time integration scheme (see Section 1.2.1), which is inherited by the `MDEngine` and `DPDEngine` classes.

Language: C#

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace AtilimGunesBaydin.Physics.ParticleSimulation
{
    public struct Vector3D
    {
        internal double X;
        internal double Y;
        internal double Z;
        public double PSize;
        public double PSizeSquared;

        public Vector3D(double x, double y, double z)
        {
            X = x;
            Y = y;
            Z = z;
            PSizeSquared = -1;
            PSize = -1;
        }

        public void SetComponents(double[] v)
        {
            X = v[0];
            Y = v[1];
            Z = v[2];
            PSizeSquared = -1;
            PSize = -1;
        }

        public void SetComponents(double x, double y, double z)
        {
            X = x;
            Y = y;
            Z = z;
            PSizeSquared = -1;
            PSize = -1;
        }

        public double Size
        {
            get
            {
                if (PSize < 0)
                    PSize = Math.Sqrt(SizeSquared);
                return PSize;
            }
        }

        public double SizeSquared
        {
            get
            {
                if (PSizeSquared < 0)
                    PSizeSquared = X * X + Y * Y + Z * Z;
                return PSizeSquared;
            }
        }

        public double[] GetComponents()
        {
            return new double[] { X, Y, Z };
        }

        public Vector3D UnitVector()
        {
```

```
            return new Vector3D(X / Size, Y / Size, Z / Size);
        }

        public void Mod(double c)
        {
            X = X % c;
            Y = Y % c;
            Z = Z % c;
            if (X < 0)
                X += c;
            if (Y < 0)
                Y += c;
            if (Z < 0)
                Z += c;
            PSize = -1;
            PSizeSquared = -1;
        }

        public Vector3D Copy()
        {
            return new Vector3D(X, Y, Z);
        }

        public void Add(Vector3D v)
        {
            X += v.X;
            Y += v.Y;
            Z += v.Z;
            PSize = -1;
            PSizeSquared = -1;
        }

        public void Subtract(Vector3D v)
        {
            X -= v.X;
            Y -= v.Y;
            Z -= v.Z;
            PSize = -1;
            PSizeSquared = -1;
        }

        public static Vector3D ZeroVector()
        {
            return new Vector3D(0, 0, 0);
        }

        public static Vector3D operator +(Vector3D v1, Vector3D v2)
        {
            return new Vector3D(v1.X + v2.X, v1.Y + v2.Y, v1.Z + v2.Z);
        }

        public static Vector3D operator -(Vector3D v1, Vector3D v2)
        {
            return new Vector3D(v1.X - v2.X, v1.Y - v2.Y, v1.Z - v2.Z);
        }

        public static Vector3D operator *(Vector3D v, double c)
        {
            return new Vector3D(c * v.X, c * v.Y, c * v.Z);
        }

        public double Dot(Vector3D v)
        {
            return X * v.X + Y * v.Y + Z * v.Z;
        }
    }


    public delegate void ProgressUpdateDelegate(double progress, int steps, double kineticEnergy,
double[][] radialDistributionFunction, double diffusionCoefficient);

    public abstract class Engine
    {
        protected int Particles;
        private double BoxSize;
        private double BoxSizeOverTwo;
        private double DeltaT;
        private double DeltaTOverTwo;
        private double DeltaTSquaredOverTwo;
        protected double RCutOff;
        protected double RCutOffSquared;

        public Vector3D[] Positions;
        protected Vector3D[] Velocities;
        private Vector3D[] Accelerations;
        public Vector3D[] Displacements;
```

```
        protected bool[,] NeighborMatrix;
        private double RNeighborSearch;
        private double RNeighborSearchSquared;
        private bool NeighborsChanged;
        private double NeighborsChangedTresholdSquared;

        private double RPotentialEnd;

        protected int DiscretizationResolution;
        protected double DiscretizationDeltaR;

        private int[] RadialDistributionBins;
        private double RadialDistributionNFactor;
        public double[][] RadialDistributionFunction;

        //private int VelocityAutocorrelationTimesteps;
        //private Vector3D[] VelocityAutocorrelationStartingVelocities;
        //public int VelocityAutocorrelationTimestep;
        //private double VelocityAutocorrelationValue;
        //private double[][] VelocityAutocorrelationFunction;

        //public double Virial;

        public int DiffusionCoefficientTimeStep;
        public Vector3D[] DiffusionCoefficientR0;
        private int DiffusionCoefficientTimeSteps;
        public double DiffusionCoefficient;

        public event ProgressUpdateDelegate ProgressUpdate;
        public bool Abort;
        public double KineticEnergy;

        public Engine(int particles, double boxSize, double deltaT, double rCutOff, double rNeighborSearch,
double rPotentialEnd, int discretizationResolution)
        {
            Particles = particles;
            BoxSize = boxSize;
            BoxSizeOverTwo = BoxSize / 2.0;
            DeltaT = deltaT;
            DeltaTOverTwo = DeltaT / 2.0;
            DeltaTSquaredOverTwo = Math.Pow(DeltaT, 2) / 2.0;
            RCutOff = rCutOff;
            RCutOffSquared = rCutOff * rCutOff;

            Positions = new Vector3D[Particles];
            Velocities = new Vector3D[Particles];
            Accelerations = new Vector3D[Particles];
            Displacements = new Vector3D[Particles];
            for (int i = 0; i < Particles; i++)
            {
                Positions[i] = Vector3D.ZeroVector();
                Velocities[i] = Vector3D.ZeroVector();
                Accelerations[i] = Vector3D.ZeroVector();
                Displacements[i] = Vector3D.ZeroVector();
            }

            RNeighborSearch = rNeighborSearch;
            RNeighborSearchSquared = rNeighborSearch * rNeighborSearch;
            NeighborMatrix = new bool[Particles, Particles];
            NeighborsChanged = true;
            NeighborsChangedTresholdSquared = Math.Pow((rNeighborSearch - rCutOff) / 2.0, 2);

            RPotentialEnd = rPotentialEnd;

            DiscretizationResolution = discretizationResolution;
            DiscretizationDeltaR = RPotentialEnd / DiscretizationResolution;

            RadialDistributionBins = new int[DiscretizationResolution];
            RadialDistributionNFactor = (2.0 * BoxSize * BoxSize * BoxSize) / (Particles * Particles * 4.0
* Math.PI * DiscretizationDeltaR);
            RadialDistributionFunction = new double[DiscretizationResolution][];
            double r = DiscretizationDeltaR / 2.0;
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                RadialDistributionFunction[i] = new double[2];
                RadialDistributionFunction[i][0] = r;
                r += DiscretizationDeltaR;
            }

            DiffusionCoefficientTimeStep = 0;
            DiffusionCoefficientTimeSteps = 500;
            DiffusionCoefficientR0 = new Vector3D[Particles];
            DiffusionCoefficient = 0;
        }

        public void VerletStep(int steps)
        {
```

```
            Vector3D DeltaPosition;

            UpdateNeighbors();
            for (int s = 0; s < steps; s++)
            {
                NeighborsChanged = false;
                for (int i = 0; i < Particles; i++)
                {
                    Velocities[i].Add(Accelerations[i] * DeltaTOverTwo);
                    DeltaPosition = (Velocities[i] * DeltaT);
                    Positions[i].Add(DeltaPosition);
                    Positions[i].Mod(BoxSize);
                    Displacements[i].Add(DeltaPosition);

                    if (!NeighborsChanged)
                        if (DeltaPosition.SizeSquared > NeighborsChangedTresholdSquared)
                            NeighborsChanged = true;
                }

                if (NeighborsChanged)
                    UpdateNeighbors();

                UpdateAccelerations();

                for (int i = 0; i < Particles; i++)
                {
                    Velocities[i].Add(Accelerations[i] * DeltaTOverTwo);
                }
            }
        }

        public void RunVerlet(int steps)
        {
            ProgressUpdate(0.0001, 0, KineticEnergy, RadialDistributionFunction, DiffusionCoefficient);
            Abort = false;

            int innersteps = 50;
            double outersteps = steps / (double)innersteps;

            for (int s = 0; s < outersteps; s++)
            {
                if (Abort)
                    break;

                VerletStep(innersteps);
                UpdateMeasurements();

                //Diffusion coefficient
                if (DiffusionCoefficientTimeStep == 0)
                {
                    for (int i = 0; i < Particles; i++)
                    {
                        DiffusionCoefficientR0[i] = Displacements[i].Copy();
                    }
                }
                else if (DiffusionCoefficientTimeStep >= DiffusionCoefficientTimeSteps)
                {
                    DiffusionCoefficient = 0;
                    for (int i = 0; i < Particles; i++)
                    {
                        DiffusionCoefficientR0[i].Subtract(Displacements[i]);
                        DiffusionCoefficient += DiffusionCoefficientR0[i].SizeSquared;
                    }
                    DiffusionCoefficient /= (Particles * 6 * DiffusionCoefficientTimeStep * DeltaT);

                    DiffusionCoefficientTimeStep = -innersteps;
                }
                DiffusionCoefficientTimeStep += innersteps;

                if (s % 2 == 0)
                    ProgressUpdate((s + 1) / outersteps, (s + 1) * innersteps, KineticEnergy,
RadialDistributionFunction, DiffusionCoefficient);
            }
        }

        public void NearestImageTransform(ref Vector3D v)
        {
            //if (Math.Abs(v.X) > BoxSizeOverTwo)
            //    v.X -= Math.Sign(v.X) * BoxSize;
            //if (Math.Abs(v.Y) > BoxSizeOverTwo)
            //    v.Y -= Math.Sign(v.Y) * BoxSize;
            //if (Math.Abs(v.Z) > BoxSizeOverTwo)
            //    v.Z -= Math.Sign(v.Z) * BoxSize;
            v.X -= BoxSize * Math.Round(v.X / BoxSize);
            v.Y -= BoxSize * Math.Round(v.Y / BoxSize);
            v.Z -= BoxSize * Math.Round(v.Z / BoxSize);
        }
```

```
        public void UpdateNeighbors()
        {
            Vector3D rij;
            bool neighbors;

            for (int i = 0; i < Particles; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    rij = Positions[i] - Positions[j];
                    NearestImageTransform(ref rij);

                    neighbors = (rij.SizeSquared < RNeighborSearchSquared);
                    NeighborMatrix[i, j] = neighbors;
                    NeighborMatrix[j, i] = neighbors;
                }
            }
        }

        public void UpdateAccelerations()
        {
            Vector3D rij;
            Vector3D force;

            //Virial = 0;

            for (int i = 0; i < Particles; i++)
                Accelerations[i] = Vector3D.ZeroVector();

            for (int i = 0; i < Particles; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    if (NeighborMatrix[i, j])
                    {
                        rij = Positions[i] - Positions[j];
                        NearestImageTransform(ref rij);

                        if (rij.SizeSquared < RCutOffSquared)
                        {
                            //Force
                            force = Force(i, j, ref rij);
                            Accelerations[i] += force;
                            Accelerations[j] -= force;

                            //Virial
                            //Virial += rij.Dot(force);
                        }
                    }
                }
            }
            //Virial /= Particles;
        }

        protected abstract Vector3D Force(int i, int j, ref Vector3D rij);

        public void UpdateMeasurements()
        {
            Vector3D rij;
            double RadialDistributionBinIndex;
            RadialDistributionBins = new int[DiscretizationResolution];

            KineticEnergy = 0;
            for (int i = 0; i < Particles; i++)
            {
                KineticEnergy += Velocities[i].SizeSquared;

                for (int j = 0; j < i; j++)
                {
                    if (NeighborMatrix[i, j])
                    {
                        rij = Positions[i] - Positions[j];
                        NearestImageTransform(ref rij);

                        //Radial distribution function
                        RadialDistributionBinIndex = (rij.Size / DiscretizationDeltaR);

                        if (RadialDistributionBinIndex < DiscretizationResolution)
                            RadialDistributionBins[(int)RadialDistributionBinIndex]++;
                    }
                }
            }

            for (int i = 0; i < DiscretizationResolution; i++)
                RadialDistributionFunction[i][1] = (RadialDistributionNFactor * RadialDistributionBins[i])
/ (RadialDistributionFunction[i][0] * RadialDistributionFunction[i][0]);
```

```
        KineticEnergy /= 2.0;
    }

    public void SetPositionsAndVelocities(double[][] positions, double[][] velocities)
    {
        for (int i = 0; i < Particles; i++)
        {
            Positions[i].SetComponents(positions[i]);
            Velocities[i].SetComponents(velocities[i]);
            Displacements[i] = Vector3D.ZeroVector();
            Accelerations[i] = Vector3D.ZeroVector();
        }

        UpdateNeighbors();
        UpdateMeasurements();
    }

    public double[][] GetPositions()
    {
        double[][] ret = new double[Particles][];

        for (int i = 0; i < Particles; i++)
        {
            ret[i] = Positions[i].GetComponents();
        }

        return ret;
    }

    public double[][] GetVelocities()
    {
        double[][] ret = new double[Particles][];

        for (int i = 0; i < Particles; i++)
        {
            ret[i] = Velocities[i].GetComponents();
        }

        return ret;
    }

    public double[][] EscapeTimeDistribution(double rmin, double rmax, double tmax, int resolution)
    {
        Vector3D rij;
        double rrange = rmax - rmin;

        int tstep = (int)((tmax / DeltaT) / (resolution - 1));

        double[][] ret = new double[resolution][];
        for (int i = 0; i < resolution; i++)
        {
            ret[i] = new double[resolution];
        }

        int[,] startrindices = new int[Particles, Particles];
        int trackedpairs = 0;
        for (int i = 0; i < Particles; i++)
        {
            for (int j = 0; j < i; j++)
            {
                rij = Positions[i] - Positions[j];
                NearestImageTransform(ref rij);

                if ((rij.Size < rmin) || (rij.Size >= rmax))
                {
                    startrindices[i, j] = -1;
                }
                else
                {
                    startrindices[i, j] = (int)(((rij.Size - rmin) / rrange) * resolution);
                    trackedpairs++;

                    ret[0][startrindices[i, j]]++;
                }
            }
        }

        for (int t = 1; t < resolution; t++)
        {
            VerletStep(tstep);

            for (int i = 0; i < Particles; i++)
            {
                for (int j = 0; j < i; j++)
                {
```

```
                        if (startrindices[i, j] != -1)
                        {
                            rij = Positions[i] - Positions[j];
                            NearestImageTransform(ref rij);

                            if (rij.Size < rmax)
                            {
                                ret[t][startrindices[i, j]]++;
                            }
                            else
                            {
                                startrindices[i, j] = -1;
                            }
                        }
                    }
                }
            }

            for (int i = resolution - 1; i >= 0; i--)
            {
                for (int j = 0; j < resolution; j++)
                {
                    ret[i][j] = (ret[i][j] / ret[0][j]);
                }
            }

            return ret;
        }


        public void testt()
        {
            ProgressUpdate(0.0001, 0, KineticEnergy, RadialDistributionFunction, DiffusionCoefficient);
        }
    }
}
```

## MD simulation interface

Code for performing MD simulations in *Mathematica.*

Language: *Mathematica*

```
(*.NET/Link*)

Needs["NETLink`"];
InstallNET[];
LoadNETAssembly["AtilimGunesBaydin.Physics.ParticleSimulation.dll"];


(*Initialization and run*)

initialize[] := Module[{},
   x = If[d == 3,
     Flatten[Table[{xx, yy, zz}, {xx, 0., l - (l/n^(1./3)),
        l/n^(1./3)}, {yy, 0., l - (l/n^(1./3)), l/n^(1./3)}, {zz, 0.,
        l - (l/n^(1./3)), l/n^(1./3)}], 2],
     Flatten[Table[{xx, yy, 0.}, {xx, 0., l - (l/Sqrt[n]),
        l/Sqrt[n]}, {yy, 0., l - (l/Sqrt[n]), l/Sqrt[n]}], 1]];
   (*x[[1]]+=l/1000.;*)
   Do[x[[i, j]] += RandomReal[{0, l/1000.}], {i, n}, {j, d}];
   v = Table[0, {n}, {3}];
   mdEngine =
    NETNew["AtilimGunesBaydin.Physics.ParticleSimulation.MDEngine", Floor[n],
     l, \[CapitalDelta]t, skin, rc, \[Epsilon], \[Sigma], 200, 1500];
   mdEngine@RunVerlet[x, v, 0];
   coreSteps = 50;
   timeStep = 1; timeStep = 0;
   time = 0;
   progress = 10^-6;
   startTime = -1;
   kineticEnergy = mdEngine@KineticEnergy[];
   potentialEnergy = mdEngine@PotentialEnergy[];
   totalEnergy = 0;
   temperature = 0;
   pressure = 0;
   vRMS = 0;
   velocityAutocorrelation = {{0, 0}};
   velocityAutocorrelationTimestep = 0;
   diffusionCoefficient = 0;
   kineticEnergyHistory = {{0, 0}};
   potentialEnergyHistory = {{0, 0}};
```

```
   totalEnergyHistory = {{0, 0}};
   pressureHistory = {{0, 0}};
   radialDistributionHistory =
    Table[mdEngine@RadialDistributionFunction, {20}];
   running = False;
   abort = False;
   startTime = -1;
   ];

runVerlet[steps_] :=
  Module[{}, running = True; startTime = AbsoluteTime[];
   startTimeStep = timeStep; Do[
    kineticEnergy = mdEngine@KineticEnergy[];
    potentialEnergy = mdEngine@PotentialEnergy[];
    totalEnergy = kineticEnergy + potentialEnergy;
    temperature = 2 kineticEnergy/(d n);
    pressure = (n temperature + mdEngine@Virial/d )/l^d;
    vRMS = Sqrt[2 kineticEnergy/n];
    velocityAutocorrelation =
     mdEngine@VelocityAutocorrelationFunction;
    velocityAutocorrelationTimestep =
     mdEngine@VelocityAutocorrelationTimestep;
    diffusionCoefficient = (\[CapitalDelta]t/d)
      Sum[velocityAutocorrelation[[j, 2]], {j,
        Length[velocityAutocorrelation]}];
    AppendTo[kineticEnergyHistory, {time, kineticEnergy}];
    AppendTo[potentialEnergyHistory, {time, potentialEnergy}];
    AppendTo[totalEnergyHistory, {time, totalEnergy}];
    AppendTo[pressureHistory, {time, pressure}];
    AppendTo[radialDistributionHistory,
     mdEngine@RadialDistributionFunction];
    If[abort, abort = False; Break[];];
    mdEngine@RunVerlet[x, v, coreSteps];
    timeStep += coreSteps;
    time = timeStep \[CapitalDelta]t;
    progress = i (coreSteps/steps);
    FinishDynamic[];,
    {i, steps/coreSteps}]; finishTime = AbsoluteTime[];
   running = False;];

(*Visualization*)

showParticles3D[] :=
  Graphics3D[{Table[{Blend[{Blue, Red}, Norm[v[[i]]]/1],
      Sphere[x[[i]], \[Sigma]/2]}, {i, n}]},
   PlotRange -> {{0, 1}, {0, 1}, {0, 1}},
   PlotRangePadding -> \[Sigma]/2, BoxRatios -> {1, 1, 1},
   Axes -> True, AxesEdge -> {{-1, -1}, {-1, -1}, {-1, -1}},
   AxesLabel -> {"x", "y", "z"}, ImageSize -> 350];

showParticles2D[] :=
  Graphics[{Table[{Blend[{Blue, Red}, Norm[v[[i]]]/1],
      Reap[
        Sow[Disk[Take[x[[i]], 2], \[Sigma]/2]];
        If[x[[i, 1]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {1, 0}, \[Sigma]/2]];
         If[x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {1, 1}, \[Sigma]/2]];];
         If[1 - x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {1, -1}, \[Sigma]/2]];];];
        If[x[[i, 2]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {0, 1}, \[Sigma]/2]];];
        If[1 - x[[i, 1]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {-1, 0}, \[Sigma]/2]];
         If[x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {-1, 1}, \[Sigma]/2]];];
         If[1 - x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {-1, -1}, \[Sigma]/2]];];];
        If[1 - x[[i, 2]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {0, -1}, \[Sigma]/2]];];
        ][[2]]}, {i, n}]}, PlotRange -> {{0, 1}, {0, 1}},
   PlotRangeClipping -> True, AspectRatio -> 1, Frame -> True,
   FrameLabel -> {"x", "y"}, ImageSize -> 350];

showParticles2DFast[] :=
  Graphics[{Table[{Blend[{Blue, Red}, Norm[v[[i]]]/1],
      Disk[Take[x[[i]], 2], \[Sigma]/2],
      Disk[Take[x[[i]], 2] + If[x[[i, 1]] < \[Sigma]/2, {1, 0}, 0] +
        If[x[[i, 2]] < \[Sigma]/2, {0, 1}, 0] +
        If[1 - x[[i, 1]] < \[Sigma]/2, {-1, 0}, 0] +
        If[1 - x[[i, 2]] < \[Sigma]/2, {0, -1}, 0], \[Sigma]/2]}, {i,
      n}]}, PlotRange -> {{0, 1}, {0, 1}}, PlotRangeClipping -> True,
   AspectRatio -> 1, Frame -> True, FrameLabel -> {"x", "y"},
   ImageSize -> 350];

manipulator[label_, {Dynamic[val_], init_}, range_,
   options___] := (val = init;
```

```
   Row[{Pane[label, 86, Alignment -> Right],
     InputField[Dynamic[val], ImageSize -> 50, options],
     Manipulator[Dynamic[val], range, ImageSize -> Tiny]},
    Spacer[1]]);

energyPlot[] :=
  Labeled[ListLinePlot[{Tooltip[kineticEnergyHistory,
      "Kinetic energy"],
     Tooltip[potentialEnergyHistory, "Potential energy"],
     Tooltip[totalEnergyHistory, "Total energy"]},
    ImageSize -> {300, 200}, AxesLabel -> {"t", ""},
    PlotRange -> All], StandardForm["Energy"]];

pressurePlot[] :=
  Labeled[ListLinePlot[pressureHistory, PlotStyle -> {Darker[Yellow]},
     ImageSize -> {300, 200}, AxesLabel -> {"t", "P"},
    PlotRange -> All, Filling -> Axis, FillingStyle -> LightYellow],
   StandardForm["Pressure"]];

radialDistributionPlot[] :=
  Labeled[ListLinePlot[(1/20)
     Plus @@ Take[radialDistributionHistory, -20],
    PlotStyle -> {Darker[Green]}, ImageSize -> {300, 200},
    AxesLabel -> {"r", "g(r)"}, PlotRange -> {{0, 1/2}, All},
    Filling -> Bottom, FillingStyle -> LightGreen],
   StandardForm["Radial distribution function"]];

velocityAutocorrelationPlot[] :=
  Labeled[ListLinePlot[velocityAutocorrelation,
    PlotStyle -> {Darker[Blue]}, ImageSize -> {300, 200},
    AxesLabel -> {"t", "C(t)"}, PlotRange -> {All, {-2, 4}},
    Filling -> Axis, FillingStyle -> LightBlue,
    Epilog ->
     Line[{{velocityAutocorrelationTimestep \[CapitalDelta]t, -4}, \
{velocityAutocorrelationTimestep \[CapitalDelta]t, 4}}]],
   StandardForm["Velocity autocorrelation"]];

(*Experiment*)

experimentPanel[] :=
  Module[{}, visualization = "Plots"; d = 3; n = 1000; rc = 2.5;
   initialize[]; Panel[Row[{
      Column[{
        Style["MD parameters", Bold, Medium],
        Row[{Spacer[90], Style["Lennard-Jones"]}],
        manipulator[
         "\[Epsilon]", {Dynamic[\[Epsilon]], 1}, {0, 2, 0.01},
         Enabled -> Dynamic[! running]],
        manipulator["\[Sigma]", {Dynamic[\[Sigma]], 1}, {0, 5, 0.01},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic[Plot[
            4 \[Epsilon] (\[Sigma]/r)^12 - (\[Sigma]/r)^6, {r, 0, rc},
             ImageSize -> 130, AspectRatio -> 1,
            PlotRange -> {{0, rc}, {-0.5, 0.5}},
            AxesLabel -> {"r",
              "\!\(\*SubscriptBox[\"\[CapitalPhi]\", \
\"lj\"]\)(r)"}]]}],
        , Style["Simulation parameters", Bold, Medium],
        manipulator[
         "\[CapitalDelta]t", {Dynamic[\[CapitalDelta]t],
          0.001}, {0.0005, 0.009, 0.0005},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic["(0.005\!\(\*SubscriptBox[\"r\", \
\"c\"]\)/\!\(\*SubscriptBox[\"v\", \"rms\"]\)=" <>
            ToString[Round[0.005/Sqrt[d], 0.0001]] <>
            ToString[")"]]}],
        manipulator[
         "\!\(\*SubscriptBox[\"r\", \"cutoff\"]\)", {Dynamic[rc],
          2.5}, {1, 10, 0.01}, Enabled -> Dynamic[! running]],
        manipulator["Skin", {Dynamic[skin], 0.2}, {0, 1, 0.1},
         Enabled -> Dynamic[! running]],
        Row[{Pane["Dimensions (d)", 86, Alignment -> Right],
          SetterBar[Dynamic[d], {2, 3},
           Enabled -> Dynamic[! running]]}, Spacer[1]],
        manipulator[
         "Box size (L)", {Dynamic[l], 8}, {Dynamic[2 rc], 40, 0.1},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic["(must be > 2\!\(\*SubscriptBox[\"r\",
RowBox[{\"cutoff\", \" \"}]]\)= " <> ToString[Round[rc 2, 0.001]] <>
            ToString[")"]]}],
        manipulator["Particles (N)", {Dynamic[n], 512},
         Dynamic[{Table[i^d, {i, Evaluate[Floor[3400^(1/d)]]}]}],
         Enabled -> Dynamic[! running]],
        Row[{Pane["\[Rho]", 86, Alignment -> Right],
```

```
          InputField[Dynamic[Round[n/l^d, 0.1]], ImageSize -> 50,
           Enabled -> False],
          Pane["(N/\!\(\*SuperscriptBox[\"L\", \"d\"]\))"]},
         Spacer[1]],
        , Style["Visualization", Bold, Medium],
        Row[{Spacer[86],

          SetterBar[
           Dynamic[visualization], {"None", "Plots", "Particles"}]},
         Spacer[1]],
        \!\(\*
GraphicsBox[
{GrayLevel[0.5], AbsoluteThickness[1], LineBox[{{0, 0.5}, {1, 0.5}}]},

ImageSize->{210., Automatic},
PlotRange->{{0, 1}, {0.49, 0.51}}]\),
        manipulator["Steps", {Dynamic[steps], 3000}, {0, 50000, 50},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Button["Initialize", initialize[];, ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Start", runVerlet[steps];, Method -> "Queued",
           ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Stop", abort = True;, Method -> "Preemptive",
           ImageSize -> 70]}],
        }, BaselinePosition -> Top],
      Column[{
        Dynamic[Refresh[Panel[Column[{Grid[{
               {"Timestep",
                " : " <> ToString[timeStep] <> " (t = " <>
                 ToString[Round[time, 0.01]] <> ")", "Pressure",
                " : " <>
                 ToString[EngineeringForm[pressure], StandardForm]},
               {"Kinetic en.",
                " : " <>
                 ToString[EngineeringForm[kineticEnergy],
                  StandardForm],
                "\!\(\*SubscriptBox[\"v\", \"rms\"]\)",
                " : " <>
                 ToString[EngineeringForm[vRMS], StandardForm]},
               {"Potential en.",
                " : " <>
                 ToString[EngineeringForm[potentialEnergy],
                  StandardForm], "Diffusion coeff.",
                " : " <>
                 ToString[EngineeringForm[diffusionCoefficient],
                  StandardForm]},
               {"Total en.",
                " : " <>
                 ToString[EngineeringForm[totalEnergy],
                  StandardForm]}}, Alignment -> Left,
              ItemSize -> {{{9, 14}}}], ,

             Switch[visualization, "None", "", "Plots",
              Column[{Row[{energyPlot[], Spacer[15],
                  radialDistributionPlot[]}],
                Row[{pressurePlot[], Spacer[15],
                  velocityAutocorrelationPlot[]}]}], "Particles",
              If[d == 2, showParticles2D[], showParticles3D[]]]
             }],
            ImageSize -> {650, Automatic},
            BaseStyle -> "StandardForm"],
           TrackedSymbols -> {timeStep, visualization}]],
        Dynamic[
         Refresh[Row[{Column[{ProgressIndicator[progress,
               ImageSize -> 150],
              ToString[Round[100. progress, 0.1]] <> "%"},
             BaseStyle -> "StandardForm"],
            If[running,
             timeLeft = (AbsoluteTime[] - startTime) (1 - progress)/
                progress;
             Column[{"Started          : " <> DateString[startTime],
               "Estimated finish : " <>
                DateString[AbsoluteTime[] + timeLeft]},
              BaseStyle -> "StandardForm"],
             If[startTime != -1,
              Column[{ToString[timeStep - startTimeStep] <>
                 " steps finished in",
                ToString[finishTime - startTime] <> " seconds."},
               BaseStyle -> "StandardForm"], ""]]}],
          TrackedSymbols -> {running, progress}]]
        }, BaselinePosition -> Top]
      }, Spacer[10]], ImageSize -> {930, Automatic}]];
```

## DPD simulation interface

Code for performing DPD simulations in *Mathematica.*

Language: *Mathematica*

```
(*.NET/Link*)

Needs["NETLink`"];
InstallNET[];
LoadNETAssembly["AtilimGunesBaydin.Physics.ParticleSimulation.dll"];

encodeNETLinkArgument[a_] :=
  a /. {\[Infinity] -> 2010.1980, -\[Infinity] -> 2010.1981};

progressUpdateHandler[p_, s_, ke_, rdf_, dcoeff_] := Module[{},
   progress = p;
   kineticEnergy = ke;
   temperature = 2 kineticEnergy/(d n);
   vRMS = Sqrt[2 kineticEnergy/n];
   diffusionCoefficient = dcoeff;
   AppendTo[kineticEnergyHistory, {time, kineticEnergy}];
   AppendTo[temperatureHistory, {time, temperature}];
   AppendTo[radialDistributionHistory, rdf];
   timeStep = s;
   time = timeStep \[CapitalDelta]t;
   FinishDynamic[];
   dpdEngine@Abort = abort;
   x = dpdEngine@GetPositions[];
   v = dpdEngine@GetVelocities[];
   ];

(*Initialization and run*)

initializeParticles[] := Module[{},
   (*x=If[d==3,RandomReal[{0,l},{n,3}],Table[{RandomReal[{0,l}],
   RandomReal[{0,l}],0},{n}]];*)
   xinit =
    If[d == 3,
     Flatten[Table[{xx, yy, zz}, {xx, 0., l - (l/n^(1./3)),
        l/n^(1./3)}, {yy, 0., l - (l/n^(1./3)), l/n^(1./3)}, {zz, 0.,
        l - (l/n^(1./3)), l/n^(1./3)}], 2],
     Flatten[Table[{xx, yy, 0.}, {xx, 0., l - (l/Sqrt[n]),
        l/Sqrt[n]}, {yy, 0., l - (l/Sqrt[n]), l/Sqrt[n]}], 1]];
   Do[xinit[[i, j]] += RandomReal[{0, rc/100.}], {i, n}, {j, d}];
   vinit = Table[0, {n}, {3}];
   ];

initialize[] := Module[{},
   dpdEngine =
    NETNew["AtilimGunesBaydin.Physics.ParticleSimulation.DPDEngineDiscretizedOmega",
      Floor[n], l, \[CapitalDelta]t, (0.277 10^-9)/(4.65 10^-10), rc,
     l/20., \[Sigma], 2000, rcmax, Length[potential]];
   AddEventHandler[dpdEngine@ProgressUpdate, progressUpdateHandler];
   x = xinit;
   v = vinit;
   dpdEngine@SetPositionsAndVelocities[x, v];
   dpdEngine@
    SetConservativeForceAndOmega[
     encodeNETLinkArgument[conservativeForce], omega];
   progress = 10^-6;
   startTime = -1;
   kineticEnergy = 0;
   temperature = 0;
   vRMS = 10^-6.;
   diffusionCoefficient = 0;
   kineticEnergyHistory = {{0, 0}};
   temperatureHistory = {{0, 0}};
   radialDistributionHistory =
    Table[dpdEngine@RadialDistributionFunction, {20}];
   running = False;
   abort = False;
   startTime = -1;
   timeStep = 1; timeStep = 0;
   time = 0;
   ];

runVerlet[steps_] :=
  Module[{}, abort = False; running = True;
   startTime = AbsoluteTime[]; timeStep = 1; timeStep = 0; time = 0;
   startTimeStep = timeStep;
   kineticEnergyHistory = {{0, 0}};
   temperatureHistory = {{0, 0}};
   radialDistributionHistory =
    Table[dpdEngine@RadialDistributionFunction, {20}];
```

```
   dpdEngine@RunVerlet[steps];
   x = dpdEngine@GetPositions[];
   v = dpdEngine@GetVelocities[];
   finishTime = AbsoluteTime[]; running = False;];

loadPotential[fileName_] :=
  Module[{},
   If[fileName =!= $Canceled,
    potentialFile =
     StringDrop[fileName, StringLength[DirectoryName[fileName]]];
    potential =
     Import[fileName, "Table"] /. {"Infinity" -> \[Infinity],
       "Inf" -> \[Infinity]};
    rcmax = potential[[-1, 1]] + ((
       potential[[2, 1]] - potential[[1, 1]])/2); rc = rcmax;
    Quiet[conservativeForce =
      forceFromPotential[potential]]; \[CapitalDelta]t =
     Round[0.005 rc/Sqrt[d], 0.001];
    omega = Table[{0, 0}, {Length[potential]}],
    potentialFile = "<None>";]];

loadOmega[fileName_] :=
  Module[{},
   If[fileName =!= $Canceled,
    omegaFile =
     StringDrop[fileName, StringLength[DirectoryName[fileName]]];
    omega = Import[fileName, "Table"] /. {"Infinity" -> \[Infinity],
       "Inf" -> \[Infinity]};, potentialFile = "<None>";]];

loadParticles[fileName_] :=
  Module[{imported},
   If[fileName =!= $Canceled, imported = Import[fileName, "Table"];
    n = imported[[1, 1]]; l = imported[[1, 2]];
    xinit = Take[Rest[imported], n]];
   vinit = Take[Rest[imported], -n];];

saveParticles[fileName_] :=
  If[fileName =!= $Canceled,
   Export[fileName, Prepend[Join[x, v], {n, l}], "Table"]];

forceFromPotential[pot_] :=
  Module[{ret},
   ret = Table[{pot[[i, 1]], -(pot[[i + 1, 2]] - pot[[i, 2]])/(
        pot[[2, 1]] - pot[[1, 1]])}, {i,
       Length[pot] - 1}] /. {Indeterminate -> \[Infinity]};
   Append[ret, {pot[[-1, 1]], ret[[-1, 2]]}] /. {\[Infinity] ->
      310}];

(*Visualization*)

showParticles3D[] :=
  Graphics3D[{Table[{Blend[{Blue, Red}, Norm[v[[i]]]/Sqrt[d]],
      Sphere[x[[i]], 0.2]}, {i, n}]},
   PlotRange -> {{0, l}, {0, l}, {0, l}}, PlotRangePadding -> 0.2,
   BoxRatios -> {1, 1, 1}, Axes -> True,
   AxesEdge -> {{-1, -1}, {-1, -1}, {-1, -1}},
   AxesLabel -> {"x", "y", "z"}, ImageSize -> 350];

showParticles2D[] :=
 Graphics[{Table[{Blend[{Blue, Red}, Norm[v[[i]]]/l],
     Reap[
       Sow[Disk[Take[x[[i]], 2], 0.2]];
       If[x[[i, 1]] < 0.2, Sow[Disk[Take[x[[i]], 2] + {l, 0}, 0.2]];
        If[x[[i, 2]] < 0.2,
         Sow[Disk[Take[x[[i]], 2] + {l, l}, 0.2]];];
        If[l - x[[i, 2]] < 0.2,
         Sow[Disk[Take[x[[i]], 2] + {l, -l}, 0.2]];];];
       If[x[[i, 2]] < 0.2,
        Sow[Disk[Take[x[[i]], 2] + {0, l}, 0.2]];];
       If[l - x[[i, 1]] < 0.2,
        Sow[Disk[Take[x[[i]], 2] + {-l, 0}, 0.2]];
        If[x[[i, 2]] < 0.2,
         Sow[Disk[Take[x[[i]], 2] + {-l, l}, 0.2]];];
        If[l - x[[i, 2]] < 0.2,
         Sow[Disk[Take[x[[i]], 2] + {-l, -l}, 0.2]];];];
       If[l - x[[i, 2]] < 0.2,
        Sow[Disk[Take[x[[i]], 2] + {0, -l}, 0.2]];];
       ][[2]]}, {i, n}]}, PlotRange -> {{0, l}, {0, l}},
  PlotRangeClipping -> True, AspectRatio -> 1, Frame -> True,
  FrameLabel -> {"x", "y"}, ImageSize -> 350]

manipulator[label_, {Dynamic[val_], init_}, range_,
   options___] := (val = init;
   Row[{Pane[label, 86, Alignment -> Right],
     InputField[Dynamic[val], ImageSize -> 50, options],
     Manipulator[Dynamic[val], range, ImageSize -> Tiny, options]},
    Spacer[1]]);
```

```
temperaturePlot[] :=
  Labeled[ListLinePlot[temperatureHistory, PlotStyle -> {Darker[Red]},
     ImageSize -> {300, 200},
    AxesLabel -> {"t", "\!\(\*SubscriptBox[\"k\", \"b\"]\)T"},
    PlotRange -> All, Filling -> Axis,
    FillingStyle -> RGBColor[1, 0.94, 0.94]],
   StandardForm["Temperature"]];

radialDistributionPlot[] :=
  Labeled[ListLinePlot[(1)
     Plus @@ Take[radialDistributionHistory, -1],
    PlotStyle -> {Darker[Green]}, ImageSize -> {300, 200},
    AxesLabel -> {"r", "g(r)"}, PlotRange -> {{0, rcmax}, All},
    Filling -> Bottom, FillingStyle -> LightGreen],
   StandardForm["Radial distribution function"]];

(*Experiment*)

experimentPanel[] :=
  Module[{}, potentialFile = "<None>"; omegaFile = "<None>";
   potential = Table[{0, 0}, {10}];
   conservativeForce = Table[{0, 0}, {10}];
   omega = Table[{0, 0}, {10}]; rcmax = 10; visualization = "Plots";
   d = 3; initializeParticles[]; initialize[]; Panel[Row[{
      Column[{
        Style["DPD parameters", Bold, Medium],
        Row[{Pane["Conserv. pot.", 86, Alignment -> Right],
          Dynamic[Labeled[
            ListLinePlot[{potential, {{rc, -5}, {rc, 5}}},
             ImageSize -> 130, AspectRatio -> 0.5,
             PlotRange -> {{0, rcmax}, {-1, 3.5}},
             PlotStyle -> {Blue, {Red, Dashed}},
             AxesLabel -> {"r", "\[Phi](r)"}], potentialFile]]},
         Spacer[1]],
        Row[{Pane["Omega", 86, Alignment -> Right],
          Dynamic[Labeled[
            ListLinePlot[{omega, {{rc, -5}, {rc, 5}}},
             ImageSize -> 130, AspectRatio -> 0.5,
             PlotRange -> {{0, rcmax}, All},
             PlotStyle -> {Purple, {Red, Dashed}},
             AxesLabel -> {"r", "\[Omega](r)"}], omegaFile]]},
         Spacer[1]],
        Row[{Pane["Resolution", 86, Alignment -> Right],
          InputField[Dynamic[Length[potential]], ImageSize -> 50,
           Enabled -> False]}, Spacer[1]],
        manipulator[
         "\[Sigma]", {Dynamic[\[Sigma]], 2.5}, {0, 30, 0.1},
         Enabled -> Dynamic[! running]],
        manipulator[
         "\!\(\*SubscriptBox[\"r\", \"cutoff\"]\)", {Dynamic[rc],
          2.5}, {(0.277 10^-9)/(4.65 10^-10), Dynamic[rcmax], 0.001},
         Enabled -> Dynamic[! running]],
        Row[{Pane["Neigh. search", 86, Alignment -> Right],
          InputField[Dynamic[Round[l/20., 0.001]], ImageSize -> 50,
           Enabled -> False],
          Pane[" + \!\(\*SubscriptBox[\"r\", \"cutoff\"]\)"]},
         Spacer[1]],
        manipulator[
         "\[CapitalDelta]t", {Dynamic[\[CapitalDelta]t],
          0.01}, {0.001, 0.1, 0.001}, Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic["(0.005\!\(\*SubscriptBox[\"r\", \
\"c\"]\)/\!\(\*SubscriptBox[\"v\", \"rms\"]\)=" <>
            ToString[Round[0.005 rc/Sqrt[d], 0.0001]] <>
            ToString[")"]]}],
        Row[{Pane["Dimensions (d)", 86, Alignment -> Right],
          SetterBar[Dynamic[d], {2, 3},
           Enabled -> Dynamic[! running]]}, Spacer[1]],
        manipulator[
         "Box size (L)", {Dynamic[l], 8}, {Dynamic[2 rc], 15, 0.1},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic["(must be > 2\!\(\*SubscriptBox[\"r\",
RowBox[{\"cutoff\", \" \"}]]\)= " <> ToString[Round[rc 2, 0.001]] <>
            ToString[")"]]}],
        manipulator["Particles (N)", {Dynamic[n], 512},
         Dynamic[{Table[i^d, {i, Evaluate[Floor[3400^(1/d)]]}]}],
         Enabled -> Dynamic[! running]],
        Row[{Pane["\[Rho]", 86, Alignment -> Right],
          InputField[Dynamic[Round[n/l^d, 0.001]], ImageSize -> 50,
           Enabled -> False],
          Pane["(N/\!\(\*SuperscriptBox[\"L\", \"d\"]\))"]},
         Spacer[1]],
        , Style["Visualization", Bold, Medium],
        Row[{Spacer[86],
          SetterBar[
```

```
          Dynamic[visualization], {"None", "Plots", "Particles"}]},
         Spacer[1]],
        \!\(\*
GraphicsBox[
{GrayLevel[0.5], AbsoluteThickness[1], LineBox[{{0, 0.5}, {1, 0.5}}]},

ImageSize->{210., Automatic},
PlotRange->{{0, 1}, {0.49, 0.51}}]\),
        manipulator["Steps", {Dynamic[steps], 500}, {0, 5000, 50},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Button["Initialize", initialize[];, ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Start", runVerlet[steps];, Method -> "Queued",
           ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Stop", abort = True;, Method -> "Preemptive",
           ImageSize -> 70]}],
        }, BaselinePosition -> Top],
      Column[{
        Row[{Button["Load potential...",
           loadPotential[
            SystemDialogInput["FileOpen",
             "Data from Johan\\Scaled\\Dimensionless\\"]]; initializeParticles[];
           initialize[];, Method -> "Queued", ImageSize -> 110],
          Spacer[5],
          Button["Load omega...",

           loadOmega[
            SystemDialogInput["FileOpen",
             "DPD\\Omega\\"]]; initializeParticles[]; initialize[];,
           Method -> "Queued", ImageSize -> 110], Spacer[5],
          Button["Create particles", initializeParticles[];
           initialize[];, Method -> "Queued", ImageSize -> 110],
          Spacer[5],
          Button["Load particles...",
           loadParticles[
            SystemDialogInput["FileOpen",
             "DPD\\Particles\\"]]; initialize[];, Method -> "Queued",
           ImageSize -> 110], Spacer[5],
          Button["Save particles...",
           saveParticles[
             SystemDialogInput["FileSave",
              "DPD\\Particles\\"]];, Method -> "Queued",
           ImageSize -> 110]}],
        Dynamic[Refresh[Panel[Column[{Grid[{
               {"Timestep",
                " : " <> ToString[timeStep] <> " (t = " <>
                 ToString[Round[time, 0.01]] <> ")",
                "\!\(\*SubscriptBox[\"v\", \"rms\"]\)",

                " : " <>
                 ToString[EngineeringForm[vRMS], StandardForm]},
               {"Kinetic en.",
                " : " <>
                 ToString[EngineeringForm[kineticEnergy],
                  StandardForm], "Diffusion coeff.",
                " : " <>
                 ToString[EngineeringForm[diffusionCoefficient],
                  StandardForm]},
               {"\!\(\*SubscriptBox[\"k\", \"b\"]\)T",
                " : " <>
                 ToString[EngineeringForm[temperature],
                  StandardForm]}
               }, Alignment -> Left, ItemSize -> {12, 1.5}], ,

             Switch[visualization, "None", "", "Plots",
              Column[{Row[{temperaturePlot[], Spacer[15],
                  radialDistributionPlot[]}], Row[{Spacer[15]}]}],
              "Particles",
              If[d == 2, showParticles2D[], showParticles3D[]]]
             }],
           ImageSize -> {650, Automatic},
           BaseStyle -> "StandardForm"],
          TrackedSymbols -> {timeStep, visualization}]],
        Dynamic[
         Refresh[Row[{Column[{ProgressIndicator[progress,
               ImageSize -> 150],
              ToString[Round[100. progress, 0.1]] <> "%"},
             BaseStyle -> "StandardForm"],
            If[running,
             timeLeft = (AbsoluteTime[] - startTime) (1 - progress)/
                progress;
             Column[{"Started          : " <> DateString[startTime],
               "Estimated finish : " <>
                DateString[AbsoluteTime[] + timeLeft]},
              BaseStyle -> "StandardForm"],
```

```
            If[startTime != -1,
             Column[{ToString[timeStep - startTimeStep] <>
                " steps finished in",
               ToString[finishTime - startTime] <> " seconds."},
              BaseStyle -> "StandardForm"], ""]]}],
          TrackedSymbols -> {running, progress}]]
       }, BaselinePosition -> Top]
     }, Spacer[10]], ImageSize -> {930, Automatic}]];
```

# Inverse Monte Carlo

## Metropolis Monte Carlo engine

The `MCEngine` class implements the Metropolis Monte Carlo algorithm (Section 1.2.2).

Language: C#

```
using System;
using System.Collections;
using System.Collections.Generic;

namespace AtilimGunesBaydin.Physics.ParticleSimulation
{
    public delegate void MCProgressUpdateDelegate(int mode, double progress, int steps, double
potentialEnergy, double[][] radialDistributionFunction);

    public class MCEngine
    {
        private int Particles;
        private double BoxSize;
        private double RCutOff;
        private double RCutOffSquared;
        private double[][] PotentialFunction;

        private Vector3D[] Positions;
        private double TotalPotential;
        private double RandomMovementRange;
        private double RandomMovementFactor;

        protected bool[,] NeighborMatrix;
        private double RNeighborSearch;
        private double RNeighborSearchSquared;

        private int DiscretizationResolution;
        private double DiscretizationDeltaR;

        private int[] RadialDistributionBins;
        private double RadialDistributionNFactor;
        private double[][] RadialDistribution;

        private Random UniformRandom;

        public event MCProgressUpdateDelegate ProgressUpdate;

        public bool Abort;

        public MCEngine(int particles, double boxSize, double rCutOff, int discretizationResolution, double
randomMovementRange, int randomSeed)
        {
            Particles = particles;
            BoxSize = boxSize;
            RCutOff = rCutOff;
            RCutOffSquared = rCutOff * rCutOff;

            Positions = new Vector3D[Particles];
            for (int i = 0; i < Particles; i++)
            {
                Positions[i] = Vector3D.ZeroVector();
            }

            UniformRandom = new Random(randomSeed);
            RandomMovementRange = randomMovementRange;
            RandomMovementFactor = Math.Sqrt(RandomMovementRange * RandomMovementRange / 3.0);

            DiscretizationResolution = discretizationResolution;
            DiscretizationDeltaR = RCutOff / DiscretizationResolution;

            PotentialFunction = new double[DiscretizationResolution][];
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                PotentialFunction[i] = new double[2];
```

```
            }

            RNeighborSearch = RCutOff + 3 * RandomMovementRange;
            RNeighborSearchSquared = RNeighborSearch * RNeighborSearch;
            NeighborMatrix = new bool[Particles, Particles];

            RadialDistributionBins = new int[DiscretizationResolution];
            RadialDistributionNFactor = (2.0 * BoxSize * BoxSize * BoxSize) / (Particles * Particles * 4.0
* Math.PI * DiscretizationDeltaR);
            RadialDistribution = new double[DiscretizationResolution][];
            double r = DiscretizationDeltaR / 2.0;
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                RadialDistribution[i] = new double[2];
                RadialDistribution[i][0] = r;
                r += DiscretizationDeltaR;
            }
        }

        private void MetropolisStep(int steps)
        {
            int randomIndex;
            Vector3D testDisplacement;
            Vector3D testPosition;
            double potential;
            double testPotential;
            double deltaPotential;

            for (int s = 0; s < steps; s++)
            {
                randomIndex = UniformRandom.Next(Particles);
                potential = NeighborsPotential(randomIndex, Positions[randomIndex]);

                testDisplacement = new Vector3D((-1 + UniformRandom.NextDouble() * 2) *
RandomMovementFactor, (-1 + UniformRandom.NextDouble() * 2) * RandomMovementFactor, (-1 +
UniformRandom.NextDouble() * 2) * RandomMovementFactor);
                testPosition = Positions[randomIndex].Copy();
                testPosition.Add(testDisplacement);
                testPotential = NeighborsPotential(randomIndex, testPosition);

                deltaPotential = testPotential - potential;
                if (deltaPotential < 0)
                {
                    Positions[randomIndex] = testPosition;
                    UpdateNeighbors(randomIndex, testDisplacement);
                }
                else if (UniformRandom.NextDouble() < Math.Exp(-deltaPotential))
                {
                    Positions[randomIndex] = testPosition;
                    UpdateNeighbors(randomIndex, testDisplacement);
                }

            }

            UpdateMeasurements();
        }

        public void SetPositionsAndPotential(double[][] positions, double[][] potentialFunction)
        {
            Vector3D rij;

            for (int i = 0; i < Particles; i++)
            {
                Positions[i].SetComponents(positions[i]);
            }

            for (int i = 0; i < DiscretizationResolution; i++)
            {
                PotentialFunction[i][0] = potentialFunction[i][0];
                PotentialFunction[i][1] = DecodeNETLinkArgument(potentialFunction[i][1]);
            }

            //Neighbor matrix
            for (int i = 0; i < Particles; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    rij = Positions[i] - Positions[j];
                    NearestImageTransform(ref rij);

                    if (rij.SizeSquared < RNeighborSearchSquared)
                    {
                        NeighborMatrix[i, j] = true;
                        NeighborMatrix[j, i] = true;
                    }
                    else
                    {
```

```
                        NeighborMatrix[i, j] = false;
                        NeighborMatrix[j, i] = false;
                    }

                }
            }

            UpdateMeasurements();
        }

        public void RunMetropolis(int steps)
        {
            ProgressUpdate(1, 0.0001, 0, PotentialEnergy, RadialDistributionFunction);
            Abort = false;

            int innersteps = 200;
            double outersteps = steps / (double)innersteps;
            for (int i = 0; i < outersteps; i++)
            {
                if (Abort)
                    break;

                MetropolisStep(innersteps);

                if (i % 10 == 0)
                    ProgressUpdate(1, (i + 1) / outersteps, (i + 1) * innersteps, PotentialEnergy,
RadialDistributionFunction);
            }

        }

        private void NearestImageTransform(ref Vector3D v)
        {
            //if (Math.Abs(v.X) > BoxSizeOverTwo)
            //    v.X -= Math.Sign(v.X) * BoxSize;
            //if (Math.Abs(v.Y) > BoxSizeOverTwo)
            //    v.Y -= Math.Sign(v.Y) * BoxSize;
            //if (Math.Abs(v.Z) > BoxSizeOverTwo)
            //    v.Z -= Math.Sign(v.Z) * BoxSize;
            v.X -= BoxSize * Math.Round(v.X / BoxSize);
            v.Y -= BoxSize * Math.Round(v.Y / BoxSize);
            v.Z -= BoxSize * Math.Round(v.Z / BoxSize);
        }

        private void UpdateNeighbors(int index, Vector3D displacement)
        {
            Vector3D rij;
            bool neighbors;

            //if (displacement.Size > ((RNeighborSearch - RCutOff) / 2))
            //{
                for (int j = 0; j < Particles; j++)
                {
                    if (j != index)
                    {
                        rij = Positions[index] - Positions[j];
                        NearestImageTransform(ref rij);

                        neighbors = (rij.SizeSquared < RNeighborSearchSquared);
                        NeighborMatrix[index, j] = neighbors;
                        NeighborMatrix[j, index] = neighbors;

                        //if (rij.SizeSquared < RNeighborSearchSquared)
                        //{
                        //    NeighborMatrix[index, j] = true;
                        //    NeighborMatrix[j, index] = true;
                        //}
                        //else
                        //{
                        //    NeighborMatrix[index, j] = false;
                        //    NeighborMatrix[j, index] = false;
                        //}

                    }
                }
            //}
        }

        private void UpdateMeasurements()
        {
            Vector3D rij;
            double RadialDistributionBinIndex;
            RadialDistributionBins = new int[DiscretizationResolution];

            TotalPotential = 0;
            for (int i = 0; i < Particles; i++)
            {
```

```
                for (int j = 0; j < i; j++)
                {
                    //if (NeighborMatrix[i, j])
                    //{
                        rij = Positions[i] - Positions[j];
                        NearestImageTransform(ref rij);

                        //Neighbor list
                        if (rij.SizeSquared < RCutOffSquared)
                        {
                            TotalPotential += PairPotential(rij.Size);
                        }

                        //Radial distribution function
                        RadialDistributionBinIndex = (rij.Size / DiscretizationDeltaR);

                        if (RadialDistributionBinIndex < DiscretizationResolution)
                            RadialDistributionBins[(int)RadialDistributionBinIndex]++;
                    //}
                }
            }

            for (int i = 0; i < DiscretizationResolution; i++)
                RadialDistribution[i][1] = (RadialDistributionNFactor * RadialDistributionBins[i]) /
(RadialDistribution[i][0] * RadialDistribution[i][0]);
        }


        private double PairPotential(double r)
        {
            double i = r / DiscretizationDeltaR;

            if (i >= DiscretizationResolution)
                return 0;

            return PotentialFunction[(int)i][1];
        }

        public double[][] RadialDistributionFunction
        {
            get
            {
                return RadialDistribution;
            }
        }

        public double PotentialEnergy
        {
            get
            {
                return TotalPotential;
            }
        }

        private double NeighborsPotential(int index, Vector3D position)
        {
            double ret = 0;
            Vector3D rij;

            for (int j = 0; j < Particles; j++)
            {
                if (NeighborMatrix[index, j])
                {
                    rij = position - Positions[j];
                    NearestImageTransform(ref rij);
                    ret += PairPotential(rij.Size);
                }
            }

            return ret;
        }

        public void InverseMonteCarloSample(int steps, ref double[] s, ref double[][] j, ref double[][]
rdf)
        {
            ProgressUpdate(2, 0.0001, 0, PotentialEnergy, RadialDistributionFunction);

            for (int a = 0; a < DiscretizationResolution; a++)
            {
                s[a] = 0;

                for (int g = 0; g < DiscretizationResolution; g++)
                {
                    j[a][g] = 0;
                }
            }
```

```
            int innersteps = 100;
            double outersteps = steps / (double)innersteps;
            for (int i = 0; i < outersteps; i++)
            {
                if (Abort)
                    break;

                MetropolisStep(innersteps);

                for (int a = 0; a < DiscretizationResolution; a++)
                {
                    s[a] += RadialDistributionBins[a];

                    for (int g = 0; g < DiscretizationResolution; g++)
                    {
                        j[a][g] += RadialDistributionBins[a] * RadialDistributionBins[g];
                    }
                }

                if (i % 10 == 0)
                    ProgressUpdate(2, (i + 1) / outersteps, (i + 1) * innersteps, PotentialEnergy,
RadialDistributionFunction);
            }

            double r = DiscretizationDeltaR / 2.0;
            for (int a = 0; a < DiscretizationResolution; a++)
            {
                s[a] /= outersteps;

                rdf[a][0] = r;
                rdf[a][1] = (RadialDistributionNFactor * s[a]) / (rdf[a][0] * rdf[a][0]);
                r += DiscretizationDeltaR;
            }

            for (int a = 0; a < DiscretizationResolution; a++)
            {
                for (int g = 0; g < DiscretizationResolution; g++)
                {
                    j[a][g] /= outersteps;
                    j[a][g] = -(j[a][g] - s[a] * s[g]);
                }
            }

        }

        private double DecodeNETLinkArgument(double a)
        {
            if (a == 2010.1980)
                return double.PositiveInfinity;
            else if (a == 2010.1981)
                return double.NegativeInfinity;
            else
                return a;
        }

        public double Test(double d)
        {
            return DecodeNETLinkArgument(d);
        }
    }
}
```

### Inverse Monte Carlo

The `MCEngine` class is called in a *Mathematica* implementation of the inverse Monte Carlo procedure (Section 3.2.1).

Language: *Mathematica*

```
(*.NET/Link*)

Needs["NETLink`"];
InstallNET[];
LoadNETAssembly["AtilimGunesBaydin.Physics.ParticleSimulation.dll"];

encodeNETLinkArgument[a_] :=
  a /. {\[Infinity] -> 2010.1980, -\[Infinity] -> 2010.1981};

progressUpdateHandler[mode_, p_, s_, pe_, rdf_] := Module[{},
   If[mode == 1,
    mceProgress = p;
```

```
    mcStep = s;,
    mcsProgress = p;
    mcStep = mcEquilibrationSteps + s;
    ];
   potentialEnergy = pe;
   AppendTo[potentialEnergyHistory, {mcStep, potentialEnergy}];
   AppendTo[mcRDFHistory, rdf];
   FinishDynamic[];
   mcEngine@Abort = abort;
   ];

(*Initialize and run*)

initialize[] := Module[{},
   x = If[d == 3,
     Flatten[Table[{xx, yy, zz}, {xx, 0., l - (l/n^(1./3)),
        l/n^(1./3)}, {yy, 0., l - (l/n^(1./3)), l/n^(1./3)}, {zz, 0.,
        l - (l/n^(1./3)), l/n^(1./3)}], 2],
     Flatten[Table[{xx, yy, 0.}, {xx, 0., l - (l/Sqrt[n]),
        l/Sqrt[n]}, {yy, 0., l - (l/Sqrt[n]), l/Sqrt[n]}], 1]];
   (*x[[1]]+=l/1000.;*)
   (*Do[x[[i,j]]+=RandomReal[{0,rc/2.}],{i,n},{j,d}];*)
   potential =
    Table[{targetRDF[[i, 1]], -Log[targetRDF[[i, 2]]]}, {i, 1,
      Length[targetRDF]}];
   (*potential=Table[{targetRDF[[i,1]],1.5-i(targetRDF[[2,1]]-
   targetRDF[[1,1]])},{i,1,Length[targetRDF]}];*)
   (*potential=Table[{targetRDF[[i,1]],If[targetRDF[[i,2]]==
   0,\[Infinity],0]},{i,Length[targetRDF]}];*)
   (*potential=tmpu;*)
   mcEngine =
    NETNew["AtilimGunesBaydin.Physics.ParticleSimulation.MCEngine", Floor[n], l, rc,
      Length[potential], rc/10, 1500];
   AddEventHandler[mcEngine@ProgressUpdate, progressUpdateHandler];
   mcEngine@
    SetPositionsAndPotential[x, encodeNETLinkArgument[potential]];
   potentialEnergy = mcEngine@PotentialEnergy;
   potentialEnergyHistory = {{0, potentialEnergy}};
   mcRDFHistory = Table[mcEngine@RadialDistributionFunction, {100}];
   imcPotentialHistory = {potential};
   imcRDFHistory = {{{0, 0}, {0, 0}}};
   imcStep = 1; imcStep = 0;
   mcStep = 1; mcStep = 0;
   imcProgress = 10^-6;
   mceProgress = 10^-6;
   mcsProgress = 10^-6;
   running = False;
   abort = False;
   imcStatus = "";
   ];

loadRDF[fileName_] :=
  Module[{},
   If[fileName =!= $Canceled,
    targetRDFFile =
     StringDrop[fileName, StringLength[DirectoryName[fileName]]];
    targetRDF = Import[fileName, "Table"];
    rc = targetRDF[[-1, 1]] + ((
       targetRDF[[2, 1]] - targetRDF[[1, 1]])/2);,
    targetRDFFile = "<None>";]];

runIMC[imcs_, mces_, mcss_, \[Lambda]_] :=
  Module[{ss, targetss, \[Delta]ss, jj, zeros, jj2,
    rdf, \[Delta]pot, \[Delta]potsmooth},
   running = True;
   ss = Table[0, {Length[targetRDF]}];
   \[Delta]ss = Table[0, {Length[targetRDF]}];
   jj = Table[0, {Length[targetRDF]}, {Length[targetRDF]}];
   rdf = Table[{0, 0}, {Length[targetRDF]}];
   Do[
    x = If[d == 3,
      Flatten[Table[{xx, yy, zz}, {xx, 0., l - (l/n^(1./3)),
         l/n^(1./3)}, {yy, 0., l - (l/n^(1./3)), l/n^(1./3)}, {zz, 0.,
          l - (l/n^(1./3)), l/n^(1./3)}], 2],
      Flatten[Table[{xx, yy, 0.}, {xx, 0., l - (l/Sqrt[n]),
         l/Sqrt[n]}, {yy, 0., l - (l/Sqrt[n]), l/Sqrt[n]}], 1]];
    mcEngine@
     SetPositionsAndPotential[x, encodeNETLinkArgument[potential]];
    potentialEnergy = mcEngine@PotentialEnergy;
    potentialEnergyHistory = {{0, potentialEnergy}};
    mcRDFHistory = Table[mcEngine@RadialDistributionFunction, {100}];
    mcStep = 0;
    mcsProgress = 10^-6;
    imcStatus = " (Equilibration)";
    runMC[mces];
    mceProgress = 1;
    If[abort, abort = False; Break[];];
```

```
    imcStatus =
     " (Sampling <\!\(\*SubscriptBox[\"S\", \"\[Alpha]\"]\)> and \
<\!\(\*SubscriptBox[\"S\", \"\[Alpha]\"]\)\!\(\*SubscriptBox[\"S\", \
\"\[Gamma]\"]\)>)";
    mcEngine@InverseMonteCarloSample[mcss, ss, jj, rdf];
    If[abort, abort = False; Break[];];
    imcStatus = " (Computing \[CapitalDelta]\[CapitalPhi])";
    targetss =
     Table[(n (n - 1))/(2 l^3)
       4 \[Pi] ((targetRDF[[\[Alpha], 1]])^2
         rc/(Length[targetRDF] - 1))
       targetRDF[[\[Alpha], 2]], {\[Alpha], Length[potential]}];
    \[Delta]ss = ss Plus @@ targetss/Plus @@ ss - targetss;
    zeros = Count[ss, 0.];
    jj2 = Table[Drop[jj[[i]], zeros], {i, zeros + 1, Length[jj]}];
    \[Delta]ss = Drop[\[Delta]ss, zeros];
    \[Delta]pot =
     Check[LinearSolve[jj2, \[Delta]ss],
      Table[0, {Length[\[Delta]ss]}]];
    \[Delta]potsmooth = smoothList[\[Delta]pot];
    \[Delta]potsmooth =
     PadLeft[\[Delta]potsmooth, Length[potential]];
    Do[potential[[i, 2]] =
      potential[[i, 2]] - \[Lambda] \[Delta]potsmooth[[i]], {i,
      Length[potential]}];
    AppendTo[imcPotentialHistory, potential];
    AppendTo[imcRDFHistory, rdf];
    imcStep++;
    imcProgress = imcStep/imcs;
    tmpss = ss;
    tmpjj = jj;
    tmprdf = rdf;
    tmptargetss = targetss;
    tmp\[Delta]ss = \[Delta]ss;
    tmpzeros = zeros;
    tmpjj2 = jj2;
    tmp\[Delta]pot = \[Delta]pot;
    tmp\[Delta]potsmooth = \[Delta]potsmooth;
    , {imcs}];
   imcStatus = "";
   running = False;
   ];

runMC[s_] :=
  Module[{}, running = True; startTime = AbsoluteTime[];
   startTimeStep = timeStep;
   mcEngine@RunMetropolis[s];
   finishTime = AbsoluteTime[]; running = False;];

Needs["Splines`"]

smoothList[list_] :=
  Module[{fit, y},
   fit[y_] =
    Fit[list, {1, y, y^2, y^3, y^4, y^5, y^6, y^7, y^8, y^9}, y];
   Table[fit[i], {i, Length[list]}]];

smoothList[list_] :=
  Module[{fit, y},
   fit = SplineFit[Table[{i, list[[i]]}, {i, Length[list]}], Bezier];
   Table[fit[i - 1][[2]], {i, Length[list]}]];

(*Visualization*)

showParticles3D[] :=
  Graphics3D[{Table[{Blue, Sphere[x[[i]], \[Sigma]/2]}, {i, n}]},
   PlotRange -> {{0, l}, {0, l}, {0, l}},
   PlotRangePadding -> \[Sigma]/2, BoxRatios -> {1, 1, 1},
   Axes -> True, AxesEdge -> {{-1, -1}, {-1, -1}, {-1, -1}},
   AxesLabel -> {"x", "y", "z"}, ImageSize -> 350];

showParticles2D[] := Graphics[{Table[{Blue,
      Reap[
        Sow[Disk[Take[x[[i]], 2], \[Sigma]/2]];
        If[x[[i, 1]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {l, 0}, \[Sigma]/2]];
         If[x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {l, l}, \[Sigma]/2]];];
         If[l - x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {l, -l}, \[Sigma]/2]];];];
        If[x[[i, 2]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {0, l}, \[Sigma]/2]];];
        If[l - x[[i, 1]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {-l, 0}, \[Sigma]/2]];
         If[x[[i, 2]] < \[Sigma]/2,
          Sow[Disk[Take[x[[i]], 2] + {-l, l}, \[Sigma]/2]];];
         If[l - x[[i, 2]] < \[Sigma]/2,
```

```
          Sow[Disk[Take[x[[i]], 2] + {-1, -1}, \[Sigma]/2]];];
        If[1 - x[[i, 2]] < \[Sigma]/2,
         Sow[Disk[Take[x[[i]], 2] + {0, -1}, \[Sigma]/2]];];
        ][[2]]}, {i, n}]}, PlotRange -> {{0, 1}, {0, 1}},
   PlotRangeClipping -> True, AspectRatio -> 1, Frame -> True,
   FrameLabel -> {"x", "y"}, ImageSize -> 350];

showParticles2DFast[] :=
  Graphics[{Table[{Blue, Disk[Take[x[[i]], 2], \[Sigma]/2],
      Disk[Take[x[[i]], 2] + If[x[[i, 1]] < \[Sigma]/2, {1, 0}, 0] +
        If[x[[i, 2]] < \[Sigma]/2, {0, 1}, 0] +
        If[1 - x[[i, 1]] < \[Sigma]/2, {-1, 0}, 0] +
        If[1 - x[[i, 2]] < \[Sigma]/2, {0, -1}, 0], \[Sigma]/2]}, {i,
      n}]}, PlotRange -> {{0, 1}, {0, 1}}, PlotRangeClipping -> True,
   AspectRatio -> 1, Frame -> True, FrameLabel -> {"x", "y"},
   ImageSize -> 350];

manipulator[label_, {Dynamic[val_], init_}, range_,
   options___] := (val = init;
   Row[{Pane[label, 86, Alignment -> Right],
     InputField[Dynamic[val], ImageSize -> 50, options],
     Manipulator[Dynamic[val], range, ImageSize -> Tiny, options]},
    Spacer[1]]);

energyPlot[] :=
  Labeled[ListLinePlot[{Tooltip[potentialEnergyHistory,
      "Hamiltonian"]}, ImageSize -> {300, 200},
    AxesLabel -> {"step", "H"}, PlotRange -> All],
   StandardForm["Hamiltonian"]];

mcRDFPlot[] :=
  Labeled[ListLinePlot[(1/20) Plus @@ Take[mcRDFHistory, -20],
    PlotStyle -> {Darker[Green]}, ImageSize -> {300, 200},
    AxesLabel -> {"r", "g(r)"}, PlotRange -> {{0, rc}, All},
    Filling -> Bottom, FillingStyle -> LightGreen],
   StandardForm["Radial distribution function"]];

imcPotentialPlot[] :=
 Labeled[ListLinePlot[imcPotentialHistory, ImageSize -> {300, 200},
   AxesLabel -> {"r", "\[CapitalPhi](r)"},
   PlotRange -> {{0, rc}, {-1, 2}},
   PlotStyle ->
    Table[If[i == Length[imcPotentialHistory], Blue,
      Blend[{LightGray, Darker[Gray]},
       i/Length[imcPotentialHistory]]], {i,
      Length[imcPotentialHistory]}]],
  StandardForm["Effective potential"]]

imcRDFPlot[] :=
 Labeled[Show[
   ListLinePlot[targetRDF, PlotStyle -> {Thick, Red},
    PlotRange -> All],
   ListLinePlot[imcRDFHistory,
    PlotStyle ->
     Table[If[i == Length[imcRDFHistory], Blue,
       Blend[{LightGray, Darker[Gray]}, i/Length[imcRDFHistory]]], {i,
       Length[imcRDFHistory]}], PlotRange -> All],
   ImageSize -> {300, 200}, AxesLabel -> {"r", "g(r)"},
   PlotRange -> {All, {0, 3}}], StandardForm["Target RDF"]]

(*Experiment*)

experimentPanel[] :=
  Module[{}, targetRDFFile = "<None>";
   targetRDF = Table[{0, 0}, {10}]; visualization = "Plots"; d = 3;
   n = 1000; rc = 2.5; labmda = 0.75; initialize[]; Panel[Row[{
      Column[{
        Style["Inverse Monte Carlo", Bold, Medium],
        Row[{Pane["Target RDF", 86, Alignment -> Right],
          Dynamic[Labeled[
            ListLinePlot[targetRDF, ImageSize -> 130,
             AspectRatio -> 0.5, PlotRange -> {{0, rc}, {0, 3.5}},
             AxesLabel -> {"r", "g(r)"}], targetRDFFile]]},
         Spacer[1]],
        Row[{Pane["Resolution", 86, Alignment -> Right],
          InputField[Dynamic[Length[potential]], ImageSize -> 50,
           Enabled -> False]}, Spacer[1]],
        Row[{Spacer[90],
          Button["Load...",
           loadRDF[
            SystemDialogInput["FileOpen",
             "Data from Johan\\Scaled\\Dimensionless\\"]]; initialize[];, Method -> "Queued",
           ImageSize -> 70]}],
        manipulator[
         "\[Lambda]", {Dynamic[lambda], 0.1}, {0, 1, 0.05}],
        , Style["Simulation parameters", Bold, Medium],
        manipulator[
```

```
         "\!\(\*SubscriptBox[\"r\", \"cutoff\"]\)", {Dynamic[rc],
          2.5}, {1, 10, 0.01}, Enabled -> False],
        Row[{Pane["Dimensions (d)", 86, Alignment -> Right],
          SetterBar[Dynamic[d], {2, 3},
           Enabled -> Dynamic[! running]]}, Spacer[1]],
        manipulator[
         "Box size (L)", {Dynamic[l], 8}, {Dynamic[2 rc], 40, 0.1},
         Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Dynamic["(must be > 2\!\(\*SubscriptBox[\"r\",
RowBox[{\"cutoff\", \" \"}]]\)= " <> ToString[Round[rc 2, 0.001]] <>
           ToString[")"]]}],
        manipulator["Particles (N)", {Dynamic[n], 512},
         Dynamic[{Table[i^d, {i, Evaluate[Floor[3400^(1/d)]]}]}],
         Enabled -> Dynamic[! running]],
        Row[{Pane["\[Rho]", 86, Alignment -> Right],
          InputField[Dynamic[Round[n/l^d, 0.001]], ImageSize -> 50,
           Enabled -> False],
          Pane["(N/\!\(\*SuperscriptBox[\"L\", \"d\"]\))"]},
         Spacer[1]],
        , Style["Visualization", Bold, Medium],
        Row[{Spacer[86],
          SetterBar[
           Dynamic[visualization], {"None", "Plots", "Particles"}]},
         Spacer[1]],
        \!\(\*
GraphicsBox[
{GrayLevel[0.5], AbsoluteThickness[1], LineBox[{{0, 0.5}, {1, 0.5}}]},

ImageSize->{210., Automatic},
PlotRange->{{0, 1}, {0.49, 0.51}}]\),
        manipulator["IMC Steps", {Dynamic[imcSteps], 3}, {0, 1000, 2},
          Enabled -> Dynamic[! running]],
        manipulator[
         "MC Equi. Steps", {Dynamic[mcEquilibrationSteps],
          300000}, {0, 50000, 50}, Enabled -> Dynamic[! running]],
        manipulator[
         "MC Smp. Steps", {Dynamic[mcSamplingSteps], 100000}, {0,
          5000, 25}, Enabled -> Dynamic[! running]],
        Row[{Spacer[90],
          Button["Initialize", initialize[];, ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Start", abort = False;
           runIMC[imcSteps, mcEquilibrationSteps, mcSamplingSteps,
            lambda];, Method -> "Queued", ImageSize -> 70]}],
        Row[{Spacer[90],
          Button["Stop", abort = True;, Method -> "Preemptive",
           ImageSize -> 70]}],
        }, BaselinePosition -> Top],
       Column[{
        Dynamic[Refresh[Panel[
           Column[{

             Grid[{{"IMC step", " : " <> ToString[imcStep]}},
              Alignment -> Left, ItemSize -> {{{9, 14}}}], ,

             Column[{Row[{imcPotentialPlot[], Spacer[15],
                 imcRDFPlot[]}], Row[{Spacer[15]}]}]
             }],
           ImageSize -> {650, Automatic},
           BaseStyle -> "StandardForm"],
          TrackedSymbols -> {imcStep, imcStatus, imcPotentialHistory,
            imcRDFHistory}]],
        Dynamic[
         Refresh[Row[{Column[{ProgressIndicator[imcProgress,
               ImageSize -> 150],
              ToString[Round[100. imcProgress, 0.1]] <> "%"},
             BaseStyle -> "StandardForm"]}],
          TrackedSymbols -> {imcProgress}]],
        Dynamic[Refresh[Panel[
           Column[{

             Grid[{{"MC step",
                " : " <> ToString[mcStep]}, {"Potential en.",
                " : " <> ToString[potentialEnergy]}},
              Alignment -> Left, ItemSize -> {{{9, 14}}}], ,

             Column[{Row[{energyPlot[], Spacer[15], mcRDFPlot[]}],
               Row[{Spacer[15]}]}]
             }],
           ImageSize -> {650, Automatic},
           BaseStyle -> "StandardForm"],
          TrackedSymbols -> {mcStep}]],
        Dynamic[
         Refresh[Row[{Column[{ProgressIndicator[mceProgress,
               ImageSize -> 150],
              ToString[Round[100. mceProgress, 0.1]] <> "%"},
```

```
            BaseStyle -> "StandardForm"],
           Column[{ProgressIndicator[mcsProgress, ImageSize -> 150],
             ToString[Round[100. mcsProgress, 0.1]] <> "%"},
            BaseStyle -> "StandardForm"],
           Column[{imcStatus}, BaseStyle -> "StandardForm"]}],
         TrackedSymbols -> {mceProgress, mcsProgress}]]
       }, BaselinePosition -> Top]
     }, Spacer[10]], ImageSize -> {930, Automatic}]];
```

# Genetic algorithms

## Base implementation

The abstract `GeneticAlgorithms` class contains the base implementation of the standard genetic algotrithm (Section 4.1.1).

Language: C#

```
using System;
using System.Drawing;
using System.Threading;
using System.Collections;
using System.Text;
using gbp.Drawing.Graph;

namespace AtilimGunesBaydin.AI.GeneticAlgorithms
{
    public abstract class GeneticAlgorithms
    {
        private Individual[] Individuals;
        private int Size;
        private int GenomeLength;
        private double GeneMinimum;
        private double GeneMaximum;

        private int Generations;
        private float CrossoverChance;
        private float MutationChance;
        private int TournamentSize;
        private float TournamentChance;
        private bool Elitism;
        private float FitnessTreshold;
        private ImprovementDelegate Improvement;
        private StoppedDelegate Stopped;
        private Graphics GStatus;
        private Bitmap GStatusBuffer;
        private Graphics GStatusBufferG;
        protected Graphics GProgress;
        protected Bitmap GProgressBuffer;
        protected Graphics GProgressBufferG;
        protected Graphics GFitness;
        protected Bitmap GFitnessBuffer;
        protected Graphics GFitnessBufferG;
        protected Color BackColor;
        protected Brush ForeColor1;
        protected Brush ForeColor2;
        private Thread T;
        private bool Abort;
        private bool ShowEval;
        private bool ShowImprove;
        protected Font f;
        private Font f2;
        private Individual AllTimeBest;
        private float AverageRawFitness;
        private float Diversity;
        protected bool IndividualsDrawn;
        private Random rnd;
        private bool[,] IdenticalMatrix;

        public GeneticAlgorithms(int size, int genomeLength, double geneMinimum, double geneMaximum, int
generations, float crossoverChance, float mutationChance, int tournamentSize, float tournamentChance, bool
elitism, float fitnessTreshold, ImprovementDelegate improvement, StoppedDelegate stopped, Graphics gStatus,
Bitmap gStatusBuffer, Graphics gProgress, Bitmap gProgressBuffer, Graphics gFitness, Bitmap gFitnessBuffer,
Color backColor, Color foreColor1, Color foreColor2)
        {
            rnd = new Random();
            Size = size;
            GenomeLength = genomeLength;
            GeneMinimum = geneMinimum;
            GeneMaximum = geneMaximum;
```

```
            Individuals = new Individual[Size];
            for (int i = 0 ; i < Size ; i++)
            {
                Individuals[i] = new Individual(GenomeLength);
                for (int n = 0 ; n < GenomeLength ; n++)
                    Individuals[i].Genome[n] = (float)(GeneMinimum + rnd.NextDouble() * (GeneMaximum -
GeneMinimum));
            }

            Generations = generations;
            CrossoverChance = crossoverChance;
            MutationChance = mutationChance;
            TournamentSize = tournamentSize;
            TournamentChance = tournamentChance;
            Elitism = elitism;
            FitnessTreshold = fitnessTreshold;
            Improvement = improvement;
            Stopped = stopped;
            GStatus = gStatus;
            GProgress = gProgress;
            GFitness = gFitness;
            GStatusBuffer = gStatusBuffer;
            GProgressBuffer = gProgressBuffer;
            GFitnessBuffer = gFitnessBuffer;
            GStatusBufferG = Graphics.FromImage(GStatusBuffer);
            GProgressBufferG = Graphics.FromImage(GProgressBuffer);
            GFitnessBufferG = Graphics.FromImage(GFitnessBuffer);
            GStatusBufferG.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
            GProgressBufferG.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
            GFitnessBufferG.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
            BackColor = backColor;
            ForeColor1 = new SolidBrush(foreColor1);
            ForeColor2 = new SolidBrush(foreColor2);
            f = new Font("Arial", 8);
            f2 = new Font("Arial", 20, FontStyle.Bold);
            ShowEval = true;
            ShowImprove = true;

            IdenticalMatrix = new bool[Size, Size];
            for (int i = 0; i < Size; i++)
                IdenticalMatrix[i, i] = true;

            AllTimeBest = new Individual(GenomeLength);
            AllTimeBest.RawFitness = 10000000;
        }

        public void SetIndividual(int index, float[] genome)
        {
            for (int n = 0; n < GenomeLength; n++)
                Individuals[index].Genome[n] = genome[n];
        }

        public void NextGeneration2()
        {
            int newindividuals = Size / 2;
            int crossovers = (int)(newindividuals * CrossoverChance / 2);
            int reproductions = newindividuals - 2 * crossovers;

            int pos = Size - newindividuals;
            for (int i = 0 ; i < crossovers ; i++)
            {
                Individual[] children = Crossover(Individuals[AGoodChromosomeIndex()],
Individuals[AGoodChromosomeIndex()]);
                Individuals[pos++] = Mutate(children[0]);
                Individuals[pos++] = Mutate(children[1]);
            }
            for (int i = 0 ; i < reproductions ; i++)
            {
                Individuals[pos++] = Mutate(Individuals[AGoodChromosomeIndex()]);
            }
        }

        public void NextGeneration()
        {
            Individual[] ng = new Individual[Size];

            int crossovers = (int)(Size * CrossoverChance / 2);
            int reproductions = Size - 2 * crossovers;

            int pos = 0;
            for (int i = 0 ; i < crossovers ; i++)
            {
                Individual[] children = Crossover(Individuals[AGoodChromosomeIndex()],
Individuals[AGoodChromosomeIndex()]);
                ng[pos++] = Mutate(children[0]);
                ng[pos++] = Mutate(children[1]);
            }
```

```
            for (int i = 0 ; i < reproductions ; i++)
            {
                ng[pos++] = Mutate(Individuals[AGoodChromosomeIndex()]);
            }

            if (Elitism)
                ng[0] = AllTimeBest;

            Individuals = ng;
        }



        private int AGoodChromosomeIndex()
        {
            int ret = rnd.Next(Size);
            int participant;
            for (int i = 0 ; i < TournamentSize - 1 ; i++)
            {
                participant = rnd.Next(Size);
                if (Individuals[participant].Fitness <= Individuals[ret].Fitness)
                    if (TournamentChance > rnd.NextDouble())
                        ret = participant;
            }
            return ret;
        }

        private Individual[] Crossover(Individual parent1, Individual parent2)
        {
            Individual[] children = new Individual[2];
            children[0] = new Individual(GenomeLength);
            children[1] = new Individual(GenomeLength);
            int cut1 = rnd.Next(GenomeLength);
            int cut2 = rnd.Next(cut1, GenomeLength);
            for (int i = 0 ; i < cut1 ; i++)
            {
                children[0].Genome[i] = parent1.Genome[i];
                children[1].Genome[i] = parent2.Genome[i];
            }
            for (int i = cut1 ; i <= cut2 ; i++)
            {
                children[0].Genome[i] = parent2.Genome[i];
                children[1].Genome[i] = parent1.Genome[i];
            }
            for (int i = cut2 + 1; i < GenomeLength ; i++)
            {
                children[0].Genome[i] = parent1.Genome[i];
                children[1].Genome[i] = parent2.Genome[i];
            }
            return children;
        }

        private Individual Mutate(Individual i)
        {
            Individual ret = new Individual(GenomeLength);
            for (int g = 0 ; g < GenomeLength ; g++)
            {
                ret.Genome[g] = i.Genome[g];
                if (rnd.NextDouble() < MutationChance)
                {
                    ret.Genome[g] *= (float)(0.5 + 1.0 * rnd.NextDouble());
                    //ret.Genome[g] = (float)(GeneMinimum + rnd.NextDouble() * (GeneMaximum -
GeneMinimum));

                    //ret.Genome[g] = (float)Math.Max(Math.Min(GeneMaximum, ret.Genome[g]), GeneMinimum);
                }
            }
            return ret;
        }

        internal bool IsEqual(float[] individual1, float[] individual2)
        {
            for (int i = 0 ; i < GenomeLength ; i++)
                if (individual1[i] != individual2[i])
                    return false;
            return true;
        }

        internal string ArrayToString(float[] a)
        {
            StringBuilder b = new StringBuilder();
            for (int i = 0; i < a.Length; i++)
            {
                b.Append(a[i]);
                if (i < a.Length - 1)
                    b.Append(", ");
            }
```

```
            return b.ToString();
        }

        public void Evolve()
        {
            Abort = false;
            T = new Thread(new ThreadStart(Run));
            T.Start();
        }

        public void Stop()
        {
            Abort = true;
        }

        public bool ShowEvaluation
        {
            get
            {
                return ShowEval;
            }
            set
            {
                ShowEval = value;
            }
        }

        public bool ShowImprovements
        {
            get
            {
                return ShowImprove;
            }
            set
            {
                ShowImprove = value;
            }
        }

        private void Run()
        {
            DateTime start = DateTime.Now;

            GStatusBufferG.Clear(BackColor);
            GStatusBufferG.DrawString("GeneticAlgorithms Environment initializing...", f, ForeColor1, 0,
0);
            GStatusBufferG.DrawString("Started " + start.ToString(), f, ForeColor2, 0, 10);
            GStatusBufferG.DrawString("Population size: " + Size.ToString(), f, ForeColor2, 0, 20);
            GStatusBufferG.DrawString("Crossover prob.: " + CrossoverChance.ToString(), f, ForeColor2, 0,
30);
            GStatusBufferG.DrawString("Mutation prob.: " + MutationChance.ToString(), f, ForeColor2, 0,
40);
            GStatus.DrawImage(GStatusBuffer, 0, 0);
            Graph2D gfitness = new Graph2D(2, GStatusBufferG, new RectangleF(240, 5, 160, 120), BackColor,
Color.DarkOrange);
            gfitness.SetComponent(0, "Average raw fitness", Color.Orange);
            gfitness.SetComponent(1, "Best raw fitness", Color.Yellow);
            GraphHistogram ghistogram = new GraphHistogram("Raw fitness distribution", 15, GStatusBufferG,
new RectangleF(420, 5, 160, 120), BackColor, Color.DarkOrange, Color.DarkOrange);

            for (int g = 0 ; g < Generations && !Abort ; g++)
            {
                GProgressBufferG.Clear(BackColor);
                GProgressBufferG.DrawString("GENERATION " + g.ToString(), f2, ForeColor1, 0, 0);
                GProgress.DrawImage(GProgressBuffer, 0, 0);

                GiveFitnesses(g);
                if (Abort)
                    break;

                GStatusBufferG.Clear(BackColor);

                GStatusBufferG.DrawString("GeneticAlgorithms Environment running...", f, ForeColor1, 0, 0);
                GStatusBufferG.DrawString("Started " + start.ToString(), f, ForeColor2, 0, 10);
                GStatusBufferG.DrawString("Population size: " + Size.ToString(), f, ForeColor2, 0, 20);
                GStatusBufferG.DrawString("Crossover prob.: " + CrossoverChance.ToString(), f, ForeColor2,
0, 30);
                GStatusBufferG.DrawString("Mutation prob.: " + MutationChance.ToString(), f, ForeColor2, 0,
40);
                GStatusBufferG.DrawString("Generation " + g.ToString() + ":", f, ForeColor1, 0, 60);
                GStatusBufferG.DrawString("Average raw fitness: " + AverageRawFitness.ToString(), f,
ForeColor2, 0, 70);
                GStatusBufferG.DrawString("Diversity: " + Diversity.ToString(), f, ForeColor2, 0, 80);
                GStatusBufferG.DrawString("Best individual", f, ForeColor1, 0, 120);
                GStatusBufferG.DrawString("Raw fitness: " + AllTimeBest.RawFitness.ToString(), f,
ForeColor2, 0, 130);
```

```
                GStatusBufferG.DrawString("Fitness: " + AllTimeBest.Fitness.ToString(), f, ForeColor2, 0,
140);

                gfitness.AddValue(0, AverageRawFitness);
                gfitness.AddValue(1, AllTimeBest.RawFitness);
                gfitness.Draw();

                float[] rawfitnesses = new float[Size];
                for (int i = 0 ; i < Size ; i++)
                    rawfitnesses[i] = Individuals[i].RawFitness;

                ghistogram.SetValues(rawfitnesses);
                ghistogram.Draw();

                GStatus.DrawImage(GStatusBuffer, 0, 0);

                //Improvement(AllTimeBest.Genome, g, Representation(Individuals[BestIndex].Genome));
                if (AllTimeBest.RawFitness <= FitnessTreshold)
                {
                    break;
                }

                NextGeneration();
            }

            GProgressBufferG.Clear(BackColor);
            GProgress.DrawImage(GProgressBuffer, 0, 0);
            Stopped();
        }

        private void GiveFitnesses(int generation)
        {
            float rawfitnessestotal = 0;
            for (int i = 0; (i < Size) && !Abort; i++)
            {
                GProgressBufferG.DrawLine(Pens.Orange, 0, 30, (225f * (i + 1) / Size), 30);
                GProgress.DrawImage(GProgressBuffer, 0, 0);

                if (!Individuals[i].FitnessGiven)
                {
                    Individuals[i].RawFitness = Fitness(Individuals[i].Genome, i, ShowEval);
                    Individuals[i].FitnessGiven = true;
                }

                rawfitnessestotal += Individuals[i].RawFitness;

                if (Individuals[i].RawFitness < AllTimeBest.RawFitness)
                {
                    AllTimeBest = Individuals[i].Copy();
                    Improvement(Individuals[i].Genome, generation, Individuals[i].RawFitness,
Representation(i));
                    if ((!ShowEval) && ShowImprove)
                        Fitness(AllTimeBest.Genome, i, true);
                }

            }
            if (Abort)
                return;

            AverageRawFitness = rawfitnessestotal / Size;
            Array.Sort(Individuals);

            bool unique;
            float uniques = 0;
            for (int i = 0 ; i < Size ; i++)
            {
                Individuals[i].Fitness = Individuals[i].RawFitness / rawfitnessestotal;

                unique = true;
                for (int p = 0 ; p < i ; p++)
                    if (IsEqual(Individuals[i].Genome, Individuals[p].Genome))
                    {
                        unique = false;
                        IdenticalMatrix[i, p] = true;
                        IdenticalMatrix[p, i] = true;
                        //break;
                    }
                if (unique)
                    uniques++;
            }
            Diversity = uniques / Size;
        }

        public abstract float Fitness(float[] individual, int index, bool showEvaluation);
        //public abstract void DrawIndividual(float[] individual, Graphics g, RectangleF position);
        public abstract object[] Representation(int index);
```

```
        private class Individual : IComparable
        {
            public float[] Genome;
            public float RawFitness;
            public float Fitness;
            public bool FitnessGiven;

            public Individual(int genomeLength)
            {
                Genome = new float[genomeLength];
                RawFitness = 0;
                Fitness = 0;
                FitnessGiven = false;
            }

            public int CompareTo(object obj)
            {
                if (obj is Individual)
                {
                    Individual i = (Individual)obj;
                    return RawFitness.CompareTo(i.RawFitness);
                }
                return -1;
            }

            public Individual Copy()
            {
                Individual ret = new Individual(Genome.Length);
                for (int i = 0; i < Genome.Length; i++)
                {
                    ret.Genome[i] = Genome[i];
                }
                ret.RawFitness = RawFitness;
                ret.Fitness = Fitness;
                ret.FitnessGiven = true;

                return ret;
            }


        }
    }

    public delegate void ImprovementDelegate(float[] solution, int generation, double fitness, object[]
representation);
    public delegate void StoppedDelegate();
}
```

## DPD

The classes `DPDGAConservative` and `DPDGADissipative` inherit the base class `GeneticAlgorithms` to define the decoding scheme and fitness evaluations for the determination of conservative and dissipative DPD interactions.

Language: C#

```
using System;
using System.Drawing;
using System.Threading;
using System.Collections;
using System.Text;
using AtilimGunesBaydin.Drawing.Graph;
using AtilimGunesBaydin.Physics.ParticleSimulation;

namespace AtilimGunesBaydin.AI.GeneticAlgorithms
{
    public class DPDGAConservative : gbp.AI.GeneticAlgorithms.GeneticAlgorithms
    {
        private DPDEngineDiscretizedOmega DPD;

        private int Particles;
        private double BoxSize;
        private double DeltaT;
        private double R0;
        private double RCutoff;
        private double Skin;
        private double Sigma;
        private double RPotentialEnd;
        private int DiscretizationResolution;
```

```
        private double[][] InitialPositions;
        private double[][] InitialVelocities;
        private double[][] Omega;
        private double[][] TargetRDF;

        private int InitSteps;
        private int SamplingSteps;

        private int DiffusionCoefficientTimeSteps;
        private int RDFZeros;

        private gbp.MathematicaLink.MathematicaLink ML;

        private double[][][] EvaluatedPotentials;
        private double[][][] EvaluatedRDFs;

        public DPDGAConservative(int size, double geneMinimum, double geneMaximum, int generations, float
crossoverChance, float mutationChance, int tournamentSize, float tournamentChance, bool elitism, float
fitnessTreshold, gbp.AI.GeneticAlgorithms.ImprovementDelegate improvement,
gbp.AI.GeneticAlgorithms.StoppedDelegate stopped, Graphics gStatus, Bitmap gStatusBuffer, Graphics
gProgress, Bitmap gProgressBuffer, Graphics gFitness, Bitmap gFitnessBuffer, int particles, double boxSize,
double deltaT, double r0, double rCutoff, double skin, double sigma, double rPotentialEnd, int
discretizationResolution, double[][] omega, double[][] targetRDF, int initSteps, int samplingSteps,
gbp.MathematicaLink.MathematicaLink ml)
            : base(size, discretizationResolution, geneMinimum, geneMaximum, generations, crossoverChance,
mutationChance, tournamentSize, tournamentChance, elitism, fitnessTreshold, improvement, stopped, gStatus,
gStatusBuffer, gProgress, gProgressBuffer, gFitness, gFitnessBuffer, Color.Black, Color.Orange,
Color.DarkOrange)
        {
            Particles = particles;
            BoxSize = boxSize;
            DeltaT = deltaT;
            R0 = r0;
            RCutoff = rCutoff;
            Skin = skin;
            Sigma = sigma;
            RPotentialEnd = rPotentialEnd;
            DiscretizationResolution = discretizationResolution;

            Omega = omega;
            TargetRDF = targetRDF;

            InitSteps = initSteps;
            SamplingSteps = samplingSteps;

            DPD = new DPDEngineDiscretizedOmega(Particles, BoxSize, DeltaT, R0, RCutoff, Skin, Sigma, 102,
RPotentialEnd, DiscretizationResolution);

            InitialPositions = new double[Particles][];
            InitialVelocities = new double[Particles][];
            for (int i = 0; i < Particles; i++)
            {
                InitialPositions[i] = new double[3];
                InitialPositions[i][0] = 0;
                InitialPositions[i][1] = 0;
                InitialPositions[i][2] = 0;

                InitialVelocities[i] = new double[3];
                InitialVelocities[i][0] = 0;
                InitialVelocities[i][1] = 0;
                InitialVelocities[i][2] = 0;
            }

            //Cubic lattice
            double dist = BoxSize / Math.Pow(Particles, 1.0 / 3.0);
            int index = 0;
            for (double x = 0; x < BoxSize; x += dist)
            {
                for (double y = 0; y < BoxSize; y += dist)
                {
                    for (double z = 0; z < BoxSize; z += dist)
                    {
                        InitialPositions[index][0] = x;
                        InitialPositions[index][1] = y;
                        InitialPositions[index][2] = z;
                        index++;
                    }
                }
            }

            DiffusionCoefficientTimeSteps = 50;
            ML = ml;

            RDFZeros = 0;
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                if (TargetRDF[i][1] == 0)
```

```
                {
                    RDFZeros++;
                }
                else
                {
                    break;
                }
            }

            EvaluatedPotentials = new double[size][][];
            EvaluatedRDFs = new double[size][][];
            for (int i = 0; i < size; i++)
            {
                EvaluatedPotentials[i] = new double[DiscretizationResolution][];
                EvaluatedRDFs[i] =  new double[DiscretizationResolution][];
                for (int j = 0; j < DiscretizationResolution; j++)
                {
                    EvaluatedPotentials[i][j] = new double[2];
                    EvaluatedRDFs[i][j] = new double[2];
                }
            }
        }

        public double[][] PotentialToForce(double[][] potential)
        {
            double[][] ret = new double[potential.Length][];

            double deltaR = potential[1][0] - potential[0][0];
            for (int i = 0; i < potential.Length - 1; i++)
            {
                ret[i] = new double[2];
                ret[i][0] = potential[i][0];
                ret[i][1] = -(potential[i + 1][1] - potential[i][1]) / deltaR;
                if (double.IsInfinity(ret[i][1]) || double.IsNaN(ret[i][1]))
                    ret[i][1] = 310;
            }
            ret[potential.Length - 1] = new double[2];
            ret[potential.Length - 1][0] = potential[potential.Length - 1][0];
            ret[potential.Length - 1][1] = ret[potential.Length - 2][1];

            return ret;
        }

        public override object[] Representation(int index)
        {
            object[] ret = new object[2];

            Bitmap cpPlot = ML.ListLinePlot(EvaluatedPotentials[index], 230, 172, "Blue", "r",
"\\[CapitalPhi]", 0, RPotentialEnd, -3, 3);
            Bitmap rdfPlot = ML.ListLinePlot(EvaluatedRDFs[index], 230, 172, "Green", "r", "RDF");

            ret[0] = cpPlot;
            ret[1] = rdfPlot;
            return ret;
        }

        private double[][] DecodePotential(float[] g)
        {
            double[][] ret = new double[DiscretizationResolution][];
            double r;

            ret[DiscretizationResolution - 1] = new double[2];
            ret[DiscretizationResolution - 1][0] = TargetRDF[DiscretizationResolution - 1][0];
            ret[DiscretizationResolution - 1][1] = 0;
            for (int i = DiscretizationResolution - 2; i >= 0; i--)
            {
                ret[i] = new double[2];
                r = TargetRDF[i][0];

                ret[i][0] = r;
                ret[i][1] = g[i];

                if (i < RDFZeros - 3)
                {
                    ret[i][1] += 100;
                }
            }

            return ML.SmoothArray(ret);
        }

        public override float Fitness(float[] g, int index, bool showEvaluation)
        {
            //
            //Decode
            //
            double[][] cp = DecodePotential(g);
```

```
            double[][] cf = PotentialToForce(cp);
            Bitmap cpPlot = null;
            Bitmap cfPlot = null;
            if (showEvaluation)
            {
                cpPlot = ML.ListLinePlot(cp, 160, 120, "Blue", "r", "\\[CapitalPhi]", 0, RPotentialEnd, -3,
3);
                cfPlot = ML.ListLinePlot(cf, 160, 120, "Gray", "r", "Fc", 0, RPotentialEnd, -3, 3);
            }

            DPD.SetPositionsAndVelocities(InitialPositions, InitialVelocities);
            DPD.SetConservativeForceAndOmega(cp, Omega);

            //
            //Initialize
            //
            int innersteps = 10;
            int outersteps = (int)((double)InitSteps / (double)innersteps);
            double[][] tempHistory = new double[outersteps][];
            for (int i = 0; i < outersteps; i++)
            {
                tempHistory[i] = new double[2];
                tempHistory[i][0] = i * innersteps * DeltaT;
            }

            for (int s = 0; s < outersteps; s++)
            {
                //if (Abort)
                //    break;
                //if (s % 2 == 0)
                DPD.UpdateMeasurements();
                tempHistory[s][0] = s * innersteps * DeltaT;
                tempHistory[s][1] = (2 * DPD.KineticEnergy) / (3 * Particles);

                if (tempHistory[s][1] > 2)
                    return 1000;

                if (showEvaluation)
                {
                    GFitnessBufferG.Clear(BackColor);
                    GFitnessBufferG.DrawString("Evaluating individual " + (index + 1).ToString(), f,
ForeColor1, 0, 0);
                    GFitnessBufferG.DrawImage(cpPlot, 0, 40);
                    GFitnessBufferG.DrawImage(cfPlot, 180, 40);
                    GFitnessBufferG.DrawLine(Pens.DarkOrange, 360, 50, 360, 150);
                    GFitnessBufferG.DrawString("Equilibration", f, ForeColor1, 380, 0);
                    GFitnessBufferG.DrawString("Time: " + (s * innersteps * DeltaT).ToString("F4"), f,
ForeColor1, 380, 10);
                    GFitnessBufferG.DrawImage(ML.ListLinePlot(DPD.RadialDistributionFunction, 160, 120,
"Green", "r", "RDF"), 380, 40);
                    GFitnessBufferG.DrawImage(ML.ListLinePlot(tempHistory, 160, 120, "Red", "t", "kBT"),
560, 40);
                    GFitnessBufferG.DrawString("kBT: " + tempHistory[s][1].ToString("F4"), f, ForeColor1,
560, 165);
                    GFitness.DrawImage(GFitnessBuffer, 0, 0);

                }

                DPD.VerletStep(innersteps);

            }

            //
            //Sampling
            //
            innersteps = 10;
            outersteps = (int)((double)SamplingSteps / (double)innersteps);

            DPD.DiffusionCoefficientTimeStep = 0;
            DPD.DiffusionCoefficient = -1;

            double[] rdf = new double[DiscretizationResolution];

            double[][] temporaryrdf = new double[DiscretizationResolution][];
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                temporaryrdf[i] = new double[2];
                temporaryrdf[i][0] = TargetRDF[i][0];
            }
            double temporaryerror = 0;

            for (int s = 0; s < outersteps; s++)
            {
                //if (Abort)
                //    break;

                //if (s % 2 == 0)
```

```
                if (showEvaluation)
                {
                    if (s != 0)
                    {
                        temporaryerror = 0;
                        for (int i = 0; i < DiscretizationResolution; i++)
                        {
                            temporaryrdf[i][1] = rdf[i] / s;
                            temporaryerror += Math.Abs(TargetRDF[i][1] - temporaryrdf[i][1]);
                        }
                        temporaryerror = Math.Pow(1 + temporaryerror, 2);
                    }

                    GFitnessBufferG.Clear(BackColor);
                    GFitnessBufferG.DrawString("Evaluating individual " + (index + 1).ToString(), f,
ForeColor1, 0, 0);
                    GFitnessBufferG.DrawImage(cpPlot, 0, 40);
                    GFitnessBufferG.DrawImage(cfPlot, 180, 40);
                    GFitnessBufferG.DrawLine(Pens.DarkOrange, 360, 50, 360, 150);
                    GFitnessBufferG.DrawString("Sampling RDF", f, ForeColor1, 380, 0);
                    GFitnessBufferG.DrawString("Time: " + (s * innersteps * DeltaT).ToString("F4"), f,
ForeColor1, 380, 10);
                    if (DPD.DiffusionCoefficient != -1)
                        GFitnessBufferG.DrawString("Diffusion coef.: " +
DPD.DiffusionCoefficient.ToString("F4"), f, ForeColor1, 380, 20);
                    GFitnessBufferG.DrawImage(ML.ListLinePlot(temporaryrdf, 160, 120, "Green", "r", "RDF"),
380, 40);
                    GFitnessBufferG.DrawString("Error: " + temporaryerror.ToString("F4"), f, ForeColor1,
380, 165);
                    GFitness.DrawImage(GFitnessBuffer, 0, 0);

                }

                DPD.VerletStep(innersteps);
                DPD.UpdateMeasurements();

                //RDF
                for (int i = 0; i < DiscretizationResolution; i++)
                {
                    rdf[i] += DPD.RadialDistributionFunction[i][1];
                }

                if (showEvaluation)
                {
                    //Diffusion coefficient
                    if (DPD.DiffusionCoefficientTimeStep == 0)
                    {
                        for (int i = 0; i < Particles; i++)
                        {
                            DPD.DiffusionCoefficientR0[i] = DPD.Displacements[i].Copy();
                        }
                    }
                    else if (DPD.DiffusionCoefficientTimeStep >= DiffusionCoefficientTimeSteps)
                    {
                        DPD.DiffusionCoefficient = 0;
                        for (int i = 0; i < Particles; i++)
                        {
                            DPD.DiffusionCoefficientR0[i].Subtract(DPD.Displacements[i]);
                            DPD.DiffusionCoefficient += DPD.DiffusionCoefficientR0[i].SizeSquared;
                        }
                        DPD.DiffusionCoefficient /= (Particles * 6 * DPD.DiffusionCoefficientTimeStep *
DeltaT);

                        DPD.DiffusionCoefficientTimeStep = -innersteps;
                    }
                    DPD.DiffusionCoefficientTimeStep += innersteps;
                }
            }

            //
            //Fitness
            //
            double error = 0;
            for (int i = 0; i < DiscretizationResolution; i++)
            {
                rdf[i] /= outersteps;
                temporaryrdf[i][1] = rdf[i];

                error += Math.Abs(TargetRDF[i][1] - rdf[i]);
            }

            EvaluatedRDFs[index] = temporaryrdf;
            EvaluatedPotentials[index] = cp;

            //error = Math.Sqrt(error);

            return (float)Math.Pow(1 + error, 2);
```

```
        }

    }

    public class DPDGADissipative : gbp.AI.GeneticAlgorithms.GeneticAlgorithms
    {
        private DPDEngineDiscretizedOmega DPD;

        private int Particles;
        private double BoxSize;
        private double DeltaT;
        private double R0;
        private double RCutoff;
        private double Skin;
        private double Sigma;
        private double RPotentialEnd;
        private int DiscretizationResolution;
        private int ETDResolution;
        private double ETDTMax;
        private double ETDRMin;
        private double ETDRMax;

        private double[][] InitialPositions;
        private double[][] InitialVelocities;
        private double[][] ConservativePotential;
        private double[][] ConservativeForce;
        private double[][] TargetETD;

        private int InitSteps;

        private int DiffusionCoefficientTimeSteps;

        private gbp.MathematicaLink.MathematicaLink ML;

        private double[][][] EvaluatedOmegas;
        private double[][][] EvaluatedETDs;

        public DPDGADissipative(int size, double geneMinimum, double geneMaximum, int generations, float
crossoverChance, float mutationChance, int tournamentSize, float tournamentChance, bool elitism, float
fitnessTreshold, gbp.AI.GeneticAlgorithms.ImprovementDelegate improvement,
gbp.AI.GeneticAlgorithms.StoppedDelegate stopped, Graphics gStatus, Bitmap gStatusBuffer, Graphics
gProgress, Bitmap gProgressBuffer, Graphics gFitness, Bitmap gFitnessBuffer, int particles, double boxSize,
double deltaT, double r0, double rCutoff, double skin, double sigma, double rPotentialEnd, int
discretizationResolution, int etdResolution, double[][] conservativePotential, double[][] targetETD, double
etdTMax, double etdRMin, double etdRMax, int initSteps, gbp.MathematicaLink.MathematicaLink ml)
            : base(size, 5, geneMinimum, geneMaximum, generations, crossoverChance, mutationChance,
tournamentSize, tournamentChance, elitism, fitnessTreshold, improvement, stopped, gStatus, gStatusBuffer,
gProgress, gProgressBuffer, gFitness, gFitnessBuffer, Color.Black, Color.Orange, Color.DarkOrange)
        {
            Particles = particles;
            BoxSize = boxSize;
            DeltaT = deltaT;
            R0 = r0;
            RCutoff = rCutoff;
            Skin = skin;
            Sigma = sigma;
            RPotentialEnd = rPotentialEnd;
            DiscretizationResolution = discretizationResolution;
            ETDResolution = etdResolution;
            ETDTMax = etdTMax;
            ETDRMin = etdRMin;
            ETDRMax = etdRMax;

            ConservativePotential = conservativePotential;
            ConservativeForce = PotentialToForce(ConservativePotential);
            TargetETD = targetETD;

            InitSteps = initSteps;

            DPD = new DPDEngineDiscretizedOmega(Particles, BoxSize, DeltaT, R0, RCutoff, Skin, Sigma, 102,
RPotentialEnd, DiscretizationResolution);

            InitialPositions = new double[Particles][];
            InitialVelocities = new double[Particles][];
            for (int i = 0; i < Particles; i++)
            {
                InitialPositions[i] = new double[3];
                InitialPositions[i][0] = 0;
                InitialPositions[i][1] = 0;
                InitialPositions[i][2] = 0;

                InitialVelocities[i] = new double[3];
                InitialVelocities[i][0] = 0;
                InitialVelocities[i][1] = 0;
                InitialVelocities[i][2] = 0;
            }
```

```
            //Cubic lattice
            double dist = BoxSize / Math.Pow(Particles, 1.0 / 3.0);
            int index = 0;
            for (double x = 0; x < BoxSize; x += dist)
            {
                for (double y = 0; y < BoxSize; y += dist)
                {
                    for (double z = 0; z < BoxSize; z += dist)
                    {
                        InitialPositions[index][0] = x;
                        InitialPositions[index][1] = y;
                        InitialPositions[index][2] = z;
                        index++;
                    }
                }
            }

            DiffusionCoefficientTimeSteps = 50;
            ML = ml;

            EvaluatedOmegas = new double[size][][];
            EvaluatedETDs = new double[size][][];
            for (int i = 0; i < size; i++)
            {
                EvaluatedOmegas[i] = new double[DiscretizationResolution][];
                for (int j = 0; j < DiscretizationResolution; j++)
                {
                    EvaluatedOmegas[i][j] = new double[2];
                }
                EvaluatedETDs[i] = new double[ETDResolution][];
                for (int j = 0; j < ETDResolution; j++)
                {
                    EvaluatedETDs[i][j] = new double[ETDResolution];
                }

            }
        }

        public double[][] PotentialToForce(double[][] potential)
        {
            double[][] ret = new double[potential.Length][];

            double deltaR = potential[1][0] - potential[0][0];
            for (int i = 0; i < potential.Length - 1; i++)
            {
                ret[i] = new double[2];
                ret[i][0] = potential[i][0];
                ret[i][1] = -(potential[i + 1][1] - potential[i][1]) / deltaR;
            }
            for (int i = potential.Length - 2; i >= 0; i--)
            {
                if (double.IsInfinity(ret[i][1]) || double.IsNaN(ret[i][1]))
                {
                    ret[i][1] = ret[i + 1][1] + 100;
                }
            }
            ret[potential.Length - 1] = new double[2];
            ret[potential.Length - 1][0] = potential[potential.Length - 1][0];
            ret[potential.Length - 1][1] = ret[potential.Length - 2][1];

            return ret;
        }

        public override object[] Representation(int index)
        {
            object[] ret = new object[2];

            Bitmap omegaPlot = ML.ListLinePlot(EvaluatedOmegas[index], 230, 172, "Purple", "r",
"\\[Omega]", 0, RPotentialEnd);
            Bitmap etdPlot = ML.ListContourPlot(EvaluatedETDs[index], 220, 220, ETDRMin, ETDRMax, 0,
ETDTMax);

            ret[0] = omegaPlot;
            ret[1] = etdPlot;
            return ret;
        }

        private double[][] DecodeOmegaOld(float[] g)
        {
            double[][] ret = new double[DiscretizationResolution][];
            double r;
            //double shift = g[0] + g[1] * RCutoff + g[2] * RCutoff * RCutoff + g[3] * Math.Pow(RCutoff,
3);
            //double scale = g[0] + g[1] * R0 + g[2] * R0 * R0 + g[3] * Math.Pow(R0, 3) - shift;

            for (int i = 0; i < DiscretizationResolution; i++)
            {
```

```
                r = ConservativePotential[i][0];

                ret[i] = new double[2];
                ret[i][0] = r;

                if ((r <= R0) || (r > RCutoff))
                {
                    ret[i][1] = 0;
                }
                else
                {
                    //ret[i][1] = Sigma * (g[0] + g[1] * r + g[2] * r * r + g[3] * Math.Pow(r, 3) - shift);
                    //ret[i][1] = Sigma * (1 - (r - R0) / (RCutoff - R0));
                    //ret[i][1] = Sigma * 8 * g[0] * (1 - Math.Pow((r - R0) / (RCutoff - R0), 2 * g[1]));
                    ret[i][1] = Sigma * 8 * g[0] * Math.Exp(-10 * g[1] * r);
                }
            }

            return ret;
        }

        private double[][] DecodeOmega(float[] g)
        {
            double[][] points = new double[5][];
            points[0] = new double[] { ConservativePotential[0][0], Sigma * g[0] };
            points[1] = new double[] { ConservativePotential[19][0], Sigma * g[1] };
            points[2] = new double[] { ConservativePotential[39][0], Sigma * g[2] };
            points[3] = new double[] { ConservativePotential[59][0], Sigma * g[3] };
            points[3] = new double[] { ConservativePotential[79][0], Sigma * g[4] };
            points[4] = new double[] { ConservativePotential[99][0], 0 };

            double[][] ret = ML.SmoothArray(points, ConservativePotential[0][0],
ConservativePotential[99][0], 100);
            double r;

            for (int i = 0; i < DiscretizationResolution; i++)
            {
                r = ConservativePotential[i][0];

                if ((r <= R0) || (r > RCutoff))
                {
                    ret[i][1] = 0;
                }
            }
            return ret;
        }

        public override float Fitness(float[] g, int index, bool showEvaluation)
        {
            string genes = ArrayToString(g);

            //
            //Decode
            //
            double[][] omega = DecodeOmega(g);
            Bitmap omegaPlot = null;
            if (showEvaluation)
            {
                omegaPlot = ML.ListLinePlot(omega, 160, 120, "Purple", "r", "\\[Omega]");
            }

            DPD.SetPositionsAndVelocities(InitialPositions, InitialVelocities);
            DPD.SetConservativeForceAndOmega(ConservativeForce, omega);

            //
            //Initialize
            //
            int innersteps = 10;
            int outersteps = (int)((double)InitSteps / (double)innersteps);
            double[][] tempHistory = new double[outersteps][];
            for (int i = 0; i < outersteps; i++)
            {
                tempHistory[i] = new double[2];
                tempHistory[i][0] = i * innersteps * DeltaT;
            }

            for (int s = 0; s < outersteps; s++)
            {
                //if (Abort)
                //    break;
                //if (s % 2 == 0)
                DPD.UpdateMeasurements();
                tempHistory[s][0] = s * innersteps * DeltaT;
                tempHistory[s][1] = (2 * DPD.KineticEnergy) / (3 * Particles);

                if (tempHistory[s][1] > 100)
                    return 1000000;
```

```
                if (showEvaluation)
                {
                    GFitnessBufferG.Clear(BackColor);
                    GFitnessBufferG.DrawString("Evaluating individual " + (index + 1).ToString() + ": " +
genes, f, ForeColor1, 0, 0);
                    GFitnessBufferG.DrawImage(omegaPlot, 0, 50);
                    GFitnessBufferG.DrawLine(Pens.DarkOrange, 260, 60, 260, 160);
                    GFitnessBufferG.DrawString("Equilibration", f, ForeColor1, 280, 10);
                    GFitnessBufferG.DrawString("Time: " + (s * innersteps * DeltaT).ToString("F4"), f,
ForeColor1, 280, 20);
                    GFitnessBufferG.DrawImage(ML.ListLinePlot(DPD.RadialDistributionFunction, 160, 120,
"Green", "r", "RDF", 0, RCutoff, 0, 3), 280, 50);
                    GFitnessBufferG.DrawImage(ML.ListLinePlot(tempHistory, 160, 120, "Red", "t", "kBT"),
460, 50);
                    GFitnessBufferG.DrawString("kBT: " + tempHistory[s][1].ToString("F4"), f, ForeColor1,
460, 175);
                    GFitness.DrawImage(GFitnessBuffer, 0, 0);
                }

                DPD.VerletStep(innersteps);

            }

            //
            //Sampling
            //
            int passes = 4;
            DPD.DiffusionCoefficientTimeStep = 0;
            DPD.DiffusionCoefficient = -1;

            double[][] etd = new double[ETDResolution][];
            double[][] temporaryetd = new double[ETDResolution][];
            for (int i = 0; i < ETDResolution; i++)
            {
                etd[i] = new double[ETDResolution];
                temporaryetd[i] = new double[ETDResolution];
            }
            double temporaryerror = 0;


            if (showEvaluation)
            {
                GFitnessBufferG.Clear(BackColor);
                GFitnessBufferG.DrawString("Evaluating individual " + (index + 1).ToString() + ": " +
genes, f, ForeColor1, 0, 0);
                GFitnessBufferG.DrawImage(omegaPlot, 0, 50);
                GFitnessBufferG.DrawLine(Pens.DarkOrange, 260, 60, 260, 160);
                GFitnessBufferG.DrawString("Sampling escape time distribution", f, ForeColor1, 280, 10);
                GFitnessBufferG.DrawString("Pass: 1", f, ForeColor1, 280, 20);
                GFitness.DrawImage(GFitnessBuffer, 0, 0);
            }
            for (int s = 0; s < passes; s++)
            {
                //if (Abort)
                //    break;

                //if (s % 2 == 0)
                if (showEvaluation)
                {
                    temporaryetd = EscapeTimeDistributionStepwise(ETDRMin, ETDRMax, ETDTMax, ETDResolution,
GFitness, new PointF(280, 50));

                    for (int i = 0; i < ETDResolution; i++)
                    {
                        for (int j = 0; j < ETDResolution; j++)
                        {
                            etd[i][j] += temporaryetd[i][j];
                            temporaryetd[i][j] = etd[i][j] / (s + 1);
                        }
                    }

                    temporaryerror = 0;
                    for (int i = 0; i < ETDResolution; i++)
                    {
                        for (int j = 0; j < ETDResolution; j++)
                        {
                            temporaryerror += Math.Pow(TargetETD[i][j] - (etd[i][j] / (s + 1)), 2);
                        }
                    }
                    temporaryerror = Math.Sqrt(temporaryerror);

                    GFitnessBufferG.Clear(BackColor);
                    GFitnessBufferG.DrawString("Evaluating individual " + (index + 1).ToString() + ": " +
genes, f, ForeColor1, 0, 0);
                    GFitnessBufferG.DrawImage(omegaPlot, 0, 50);
                    GFitnessBufferG.DrawLine(Pens.DarkOrange, 260, 60, 260, 160);
```

```
                    GFitnessBufferG.DrawString("Sampling escape time distribution", f, ForeColor1, 280,
10);
                    GFitnessBufferG.DrawString("Pass: " + (s + 2).ToString(), f, ForeColor1, 280, 20);
                    if (DPD.DiffusionCoefficient != -1)
                        GFitnessBufferG.DrawString("Diffusion coef.: " +
DPD.DiffusionCoefficient.ToString("F4"), f, ForeColor1, 280, 30);
                    //GFitnessBufferG.DrawImage(ML.ArrayPlot(etd, 160, 120, ETDRMin, ETDRMax, 0, ETDTMax),
380, 40);
                    GFitnessBufferG.DrawImage(ML.ArrayPlot(temporaryetd, 150, 150, ETDRMin, ETDRMax, 0,
ETDTMax), 440, 50);
                    GFitnessBufferG.DrawString("Error: " + temporaryerror.ToString("F4"), f, ForeColor1,
600, 50);
                    GFitness.DrawImage(GFitnessBuffer, 0, 0);

                    //Diffusion coefficient
                    if (DPD.DiffusionCoefficientTimeStep == 0)
                    {
                        for (int i = 0; i < Particles; i++)
                        {
                            DPD.DiffusionCoefficientR0[i] = DPD.Displacements[i].Copy();
                        }
                    }
                    else if (DPD.DiffusionCoefficientTimeStep >= DiffusionCoefficientTimeSteps)
                    {
                        DPD.DiffusionCoefficient = 0;
                        for (int i = 0; i < Particles; i++)
                        {
                            DPD.DiffusionCoefficientR0[i].Subtract(DPD.Displacements[i]);
                            DPD.DiffusionCoefficient += DPD.DiffusionCoefficientR0[i].SizeSquared;
                        }
                        DPD.DiffusionCoefficient /= (Particles * 6 * DPD.DiffusionCoefficientTimeStep *
DeltaT);

                        DPD.DiffusionCoefficientTimeStep = -(int)(ETDTMax / DeltaT);
                    }
                    DPD.DiffusionCoefficientTimeStep += (int)(ETDTMax / DeltaT);
                }
                else
                {
                    temporaryetd = DPD.EscapeTimeDistribution(ETDRMin, ETDRMax, ETDTMax, ETDResolution);

                    for (int i = 0; i < ETDResolution; i++)
                    {
                        for (int j = 0; j < ETDResolution; j++)
                        {
                            etd[i][j] += temporaryetd[i][j];
                        }
                    }
                }
            }

            //
            //Fitness
            //
            double error = 0;
            for (int i = 0; i < ETDResolution; i++)
            {
                for (int j = 0; j < ETDResolution; j++)
                {
                    etd[i][j] /= (double)passes;
                    error += Math.Pow(TargetETD[i][j] - etd[i][j], 2);
                }
            }

            EvaluatedETDs[index] = etd;
            EvaluatedOmegas[index] = omega;

            return (float)Math.Sqrt(error);
        }

        public double[][] EscapeTimeDistributionStepwise(double rmin, double rmax, double tmax, int
resolution, Graphics g, PointF pos)
        {
            Vector3D rij;
            double rrange = rmax - rmin;

            int tstep = (int)((tmax / DeltaT) / (resolution - 1));

            double[][] ret = new double[resolution][];
            for (int i = 0; i < resolution; i++)
            {
                ret[i] = new double[resolution];
                for (int j = 0; j < resolution; j++)
                {
                    ret[i][j] = 1;
                }
            }
```

```
            double[] t0particles = new double[resolution];

            int[,] startrindices = new int[Particles, Particles];
            int trackedpairs = 0;
            for (int i = 0; i < Particles; i++)
            {
                for (int j = 0; j < i; j++)
                {
                    rij = DPD.Positions[i] - DPD.Positions[j];
                    DPD.NearestImageTransform(ref rij);

                    if ((rij.Size < rmin) || (rij.Size >= rmax))
                    {
                        startrindices[i, j] = -1;
                    }
                    else
                    {
                        startrindices[i, j] = (int)(((rij.Size - rmin) / rrange) * resolution);
                        trackedpairs++;

                        ret[0][startrindices[i, j]] = 1;
                        t0particles[startrindices[i, j]]++;
                    }
                }
            }

            for (int t = 1; t < resolution; t++)
            {
                DPD.VerletStep(tstep);

                for (int i = 0; i < Particles; i++)
                {
                    for (int j = 0; j < i; j++)
                    {
                        if (startrindices[i, j] != -1)
                        {
                            rij = DPD.Positions[i] - DPD.Positions[j];
                            DPD.NearestImageTransform(ref rij);

                            if (rij.Size < rmax)
                            {
                                ret[t][startrindices[i, j]]++;
                            }
                            else
                            {
                                startrindices[i, j] = -1;
                            }
                        }
                    }
                }

                for (int i = 0; i < resolution; i++)
                {
                    ret[t][i] /= t0particles[i];
                }

                if (t % 2 == 0)
                {
                    g.DrawImage(ML.ArrayPlot(ret, 150, 150, ETDRMin, ETDRMax, 0, ETDTMax), pos);
                }
            }

            return ret;
        }
    }
}
```

### Auxiliary Mathematica code

The *Mathematica* code for fitting Bézier splines to random walks (Section 4.3.1) and producing plots is wrapped within the `MathematicaLink` class as strings that are evaluated through `Wolfram.NETLink.IKernelLink` interface.

Language: C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Drawing;
using Wolfram.NETLink;
using Wolfram.NETLink.UI;

namespace AtilimGunesBaydin.MathematicaLink
{
    public class MathematicaLink
    {
        IKernelLink Link;

        public MathematicaLink()
        {
            Link = MathLinkFactory.CreateKernelLink();
            Link.WaitAndDiscardAnswer();

            //Function definitions
            Link.Evaluate("Needs[\"Splines`\"]");
            Link.WaitAndDiscardAnswer();
            Link.Evaluate("smoothList[list_]:=Module[{fit,y},fit=SplineFit[list,Bezier];Table[fit[i-
1],{i,Length[list]}]]");
            Link.WaitAndDiscardAnswer();
            Link.Evaluate("smoothList[points_, rmin_, rmax_,
resolution_]:=Module[{fit},fit=SplineFit[points,Bezier];Table[{rmin+x(rmax-rmin)/(Length[points]-
1),fit[x][[2]]},{x,0.,Length[points]-1.,(Length[points]-1)/(resolution-1.)}]]");
            Link.WaitAndDiscardAnswer();
        }

        public string EncodeExpression(double[][] array)
        {
            StringBuilder s = new StringBuilder();
            s.Append('{');
            string test;
            for (int i = 0; i < array.Length; i++)
            {
                s.Append('{');
                for (int j = 0; j < array[0].Length; j++)
                {
                    test = array[i][j].ToString();
                    test = test.Replace("E", "*10^");
                    s.Append(test);
                    if (j != array[0].Length - 1)
                    {
                        s.Append(',');
                    }
                }
                s.Append('}');
                if (i != array.Length - 1)
                {
                    s.Append(',');
                }
            }
            s.Append('}');

            return s.ToString();
        }

        public Bitmap ListLinePlot(double[][] list, int width, int height, string color, string xlabel,
string ylabel)
        {
            string options = ",PlotRange->{{0,All},Automatic},Background->Black,AxesStyle-
>Orange,AspectRatio->0.75,PlotStyle->{" + color + ",Thick},AxesLabel->{\"" + xlabel + "\",\"" + ylabel +
"\"},ImageSize->{" + width.ToString() + "," + height.ToString() + "}";
            string expr = "ListLinePlot[" + EncodeExpression(list) + options + "]";
            return (Bitmap)Link.EvaluateToImage(expr, width, height);
        }

        public Bitmap ListLinePlot(double[][] list, int width, int height, string color, string xlabel,
string ylabel, double xmin, double xmax)
        {
            string options = ",PlotRange->{{" + xmin + "," + xmax + "},All},Background->Black,AxesStyle-
>Orange,AspectRatio->0.75,PlotStyle->{" + color + ",Thick},AxesLabel->{\"" + xlabel + "\",\"" + ylabel +
"\"},ImageSize->{" + width.ToString() + "," + height.ToString() + "}";
            string expr = "ListLinePlot[" + EncodeExpression(list) + options + "]";
            return (Bitmap)Link.EvaluateToImage(expr, width, height);
        }

        public Bitmap ListLinePlot(double[][] list, int width, int height, string color, string xlabel,
string ylabel, double xmin, double xmax, double ymin, double ymax)
        {
            string options = ",PlotRange->{{" + xmin + "," + xmax + "},{" + ymin + "," + ymax +
"}},Background->Black,AxesStyle->Orange,AspectRatio->0.75,PlotStyle->{" + color + ",Thick},AxesLabel->{\""
+ xlabel + "\",\"" + ylabel + "\"},ImageSize->{" + width.ToString() + "," + height.ToString() + "}";
            string expr = "ListLinePlot[" + EncodeExpression(list) + options + "]";
            return (Bitmap)Link.EvaluateToImage(expr, width, height);
        }
```

```
        public Bitmap ArrayPlot(double[][] list, int width, int height, double rangexmin, double rangexmax,
double rangeymin, double rangeymax)
        {
            string expr = "ArrayPlot[Reverse[" + EncodeExpression(list) + "],Background->Black,FrameStyle-
>Orange,FrameTicks->{{All,None},{All,None}},DataRange->{{" + rangexmin.ToString() + "," +
rangexmax.ToString() + "},{" + rangeymin.ToString() + "," + rangeymax.ToString() + "}},AspectRatio-
>1,ImageSize->{" + width.ToString() + "," + height.ToString() + "}]";
            return (Bitmap)Link.EvaluateToImage(expr, width, height);
        }

        public Bitmap ListContourPlot(double[][] list, int width, int height, double rangexmin, double
rangexmax, double rangeymin, double rangeymax)
        {
            string expr = "ListContourPlot[" + EncodeExpression(list) + ",Background->Black,FrameStyle-
>Orange,AspectRatio->1,Contours->15,ColorFunction->(GrayLevel[1-#]&),ImageSize->{" + width.ToString() + ","
+ height.ToString() + "},DataRange->{{" + rangexmin.ToString() + "," + rangexmax.ToString() + "},{" +
rangeymin.ToString() + "," + rangeymax.ToString() + "}}]";
            return (Bitmap)Link.EvaluateToImage(expr, width, height);
        }

        public double[][] SmoothArray(double[][] array)
        {
            Link.Evaluate("smoothList[" + EncodeExpression(array) + "]");
            Link.WaitForAnswer();
            double[,] tmp = (double[,])Link.GetArray(typeof(double), 2);
            double[][] ret = new double[array.Length][];
            for (int i = 0; i < array.Length; i++)
            {
                ret[i] = new double[array[0].Length];
                for (int j = 0; j < array[0].Length; j++)
                {
                    ret[i][j] = tmp[i, j];
                }
            }

            return ret;
        }

        public double[][] SmoothArray(double[][] points, double rmin, double rmax, int resolution)
        {
            Link.Evaluate("smoothList[" + EncodeExpression(points) + "," + rmin.ToString() + "," +
rmax.ToString() + "," + resolution.ToString() + "]");
            Link.WaitForAnswer();
            double[,] tmp = (double[,])Link.GetArray(typeof(double), 2);
            double[][] ret = new double[resolution][];
            for (int i = 0; i < resolution; i++)
            {
                ret[i] = new double[2];
                ret[i][0] = tmp[i, 0];
                ret[i][1] = tmp[i, 1];
            }

            return ret;
        }

        public void Close()
        {
            Link.Close();
        }
    }
}
```