WATER CONFINED IN NANOPOROUS SILICA

by

Filip Sund

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2014

Abstract

In this thesis we have studied the structure and transport properties of water trapped in nanoporous fractures in silica. The studies are performed using a numerical method called molecular dynamics, using an advanced, accurate, and verified model of silica and water, based on quantum mechanical and experimental studies of these systems.

We have developed a method for generating random fractures which are statistically similar to naturally occurring fractures, which is used to create nanoporous silica. This method is validated using a fractal-based characterization method. We have also developed a method for filling the fractures and pores with water

We have studied systems of silica with nanoscale fractures of sizes ranging from 14 to 80 Ångström, with a Hurst (or roughness) exponent of approximately 0.75. We have found that approach we used for filling the fractures worked well in large fractures, but that the method breaks down in rough fractures smaller than around 20 Ångströmin one or more dimensions.

We have also found that the structure and transport properties of water is bulk-like down to around 8 Å from a the silica surface. We found that in a region around 7 Å and closer to the silica surface, the water density was noticeably reduced, and the self-diffusion of water was also reduced. Upon studies of the intermolecular structure of water we conclude that the interactions between silica and water causes changes in the water structure, which again leads to changes in the density and diffusive properties of water near the silica surface.

We also found that the actual geometry and structure of the pore or fracture had little to no effect on the properties of water in the systems we studied, with the only real deviation from this was found in a 14.4 Å wide fracture, which had some differences in diffusion and structure near the silica matrix compared to the other systems.

Acknowledgements

This master thesis was written at the University of Oslo, with supervision from Professors Anders Malthe-Sørenssen and Morten Hjort-Jensen, finishing in June 2014.

I initially started my master studies with the goal of doing experimental studies of semiconductors and solar cells, but after attending Morten's course in computational physics, and with some careful nudging from Anders Hafreager, I realized that computational physics was a field that I found much more interesting. After this discovery I quickly changed the direction of my master studies, and ended up at the Computational Physics branch of the Physics department.

I want to thank Anders Malthe-Sørenssen for excellent guidance during my work, and Morten Hjort-Jensen for his supervision. I also want to thank Anders Hafreager, Mathilde Kamperud, Jørgen Trømborg, Jørgen Høgberget, and all others at Computational Physics at the University for many good discussions and talks that made this thesis into what it is.

I would also like to thank the developers of Ovito[40], Inkscape[42] and Matplotlib for creating such excellent software, for freely distributing the programs, and for making the source code freely available using an open source software license. These programs have been used for creating nearly all graphics and illustrations presented in this thesis.

Lastly, I thank my wonderful wife Silje for keeping me motivated and energized throughout my master studies, and for feeding me with cake.

Contents

Introduction to the thesis

Background and motivation

The behaviour of water in nanoporous silica is important for many geological, biological, physical and biomedical processes. We are able to produce nanoporous silica with tuneable pore sizes, and the pore surfaces can be functionalized, which can lead to a wide range of industrial applications, many of which we have yet to discover, or the process behind is not fully understood. Examples of such applications are filtration, catalysis, phase separation, medical delivery systems, desalination of sea water, and CO_2 capture and storage.

Experiments have shown that the behaviour of fluids confined in nanoscale pores is different to that of bulk fluids. This is caused not only by the confinement to a small volume, but also by the high ratio of surface area to pore volume. There is evidence that water near surfaces can reach super-cooled states without forming ice, and that the density and dynamics of water near surfaces is different than that of bulk water. The detailed mechanisms for such behaviours are still uncertain.

While there has been performed significant number of high-quality experimental studies of the properties of water in nanoscale pores in the recent years, simulational studies are still limited, and many of the simulational studies that have been performed have been for small systems with limited dynamics. This makes this a good area for application of large-scale modelling and simulation.

We choose to model the nanoporous system at an atomic scale, both because we think that such a scale is appropriate for predicting the effects at nano-scales, and because similar studies have been performed with good results. We choose to use molecular dynamics to to the modelling, but an alternative method is direct simulation Monte Carlo, which have shown equally promising results applied on similar problems.

The goal of this master thesis is to do a numerical study of water trapped in nanoscale pores in solid silica, using a well-constrained interatomic potential

that accurately predict the behaviour of both silica, water, and the interactions between the two. The potential we use has been developed by researchers at the University of Southern California, and the models have been well tested and calibrated to experimental and other numerical studies.

To study water trapped in nanoporous silica we need to develop reproducable methods for intializing realistic nanoporous silica, and methods for filling the nanoscale pores with water. We want to study the structure, behaviour and transport properties of water in silica, so we also need to find some appropriate way of quantifying and analyzing the dynamic and structural properties of water near the surface.

Structure of the thesis

We start with a description of the numerical methods we will use to study silica and water. We then describe the methods we use for creating and initializing the systems we want to simulate and do measurements on. We then prepare, simulate, and do measurements on the systems. Finally we analyze the results, compare the measurements against each other, and draw some conclusions.

Part I

Molecular dynamics

Introduction

In this chapter we will give an overview of how simulations of atomic systems are done using molecular dynamics. We will show the theory that makes molecular dynamics so efficient and powerful, and we will show how to build up and implement a molecular dynamics program.

The program used for producing the results presented later in this thesis uses a much more sophisticated program and model for the interactions between the atoms than the one presented in chapters 1 and 2. We will nevertheless gain a lot of insight into this program by starting with a a simpler case.

To do an exact study of a many-body atomic system like water and silica, we have to take into account the quantum mechanical nature of the atoms and molecules in the system. An oxygen molecule consists of 8 electrons, 8 protons and 8 neutrons, all interacting with each other, and each with 3 translational degrees of freedom. This makes doing calculations on something that might appear simple, pretty complex if we want to do it properly. If we want to study a system consisting of more than a couple of oxygen atoms we see that the number of particles and degrees of freedom quickly makes the problem grow to intractable proportions. Since we are mainly interested in the equilibrium and transport properties of the system, we can reduce the problem to something we can handle by using results from underlying quantum mechanical calculations, and experimental studies, to develop approximate models of the system. We do this by assuming that the many-body system behaves clasically, and model all atoms as point particles. We further create potentials from quantum mechanical results that approximate the exact forces between the atoms, and adjust the parameters of these potentials to fit experimental and computational results. These potentials only depend on the positions of the atoms we want to simulate, which is orders of magnitude faster to evaluate than calculating the exact forces between the atoms from quantum mechanical principles. We then solve Newton's equations of motion to evolve the system in time.

After presenting a simple molecular dynamics model we briefly mention some methods for optimizing such a program, before we look at what we are able to

study using a molecular dynamics simulation. We first look at how to measure the temperature and pressure in a system, before we look at how to study different ensembles using so-called *thermostats*, which can control the temperature, pressure, or other physical proterties of a system. We conclude part I with an overview of the program used for producing the results presented later in this thesis, which we use to model water, silica, and the interactions between the two.

Chapter 1

A simple molecular dynamics model

In molecular dynamics we study systems of many interacting atoms and molecules by assuming that they behave classically, and solve Newton's equations of motion using an appropriate integration scheme to evolve the system in time. By assuming that the atoms behave classically we mean that we model the atoms as point particles, and characterize them using their position, $\boldsymbol{r}$, velocity, $\boldsymbol{v}$ and the force acting on them, $\boldsymbol{F}$. The interactions between the atoms are described using potentials. It is into these potentials we bake the physical insight of the systems we want to simulate, which we often do by finding potentials using studies and simulations of the underlying quantum mechanical nature of the interactions between the atoms in the system, and also by comparing with results from experimental studies.

1.1 The main program

Using a molecular dynamic simulations we can start from any initial state S_0 and evolve this state in time. We can stop the simulations at any time, and continue the simulations from any saved state. This is a powerful tool that can for example be used to study different variations of a system, using the same initial conditions.

Most molecular dynamics programs will follow a flow similar to the following procedure:

- Initialize the system by setting up the initial positions and velocities for all atoms. This is usually in one of two ways

    - Load a saved state from a previous simulation
    - Generate positions and velocities randomly, or following some rules to control the physical properties of the system. When generating random velocities we usually remove any net velocity, to avoid drift.
- For each timestep
    - Calculate the forces between the atoms.
    - Integrate Newton's equations of motion, using an appropriate integration scheme.
    - Sample the values of the quantities we want to study, and add to the averages.
- After all timesteps have been finished we print out the measured quantities, and we could also save the state of the system so we can continue from this state later.

An example of a program that implements the above procedure can be seen in listing 1.1.

```
System system = initializeSystem(parameters);
double time = 0.0;  // initial time
double dt = 0.01;   // timestep
for (double time = 0; time < tMax; time += dt) {
    calculateForces(system);
    integrateEquationsOfMotion(system, dt);
    sample(system);
}
```

Listing 1.1: An example of a typical implementation of a molecular dynamics program using object-oriented programming. See listings 1.2, 1.4 and 1.9 for examples of implementations of the functions `calculateForces`, `integrateEquationsOfMotion`, and `sample`.

When starting a new simulation we usually initialize the positions of the atoms by putting them on a regular grid, like a face-centered cubic (fcc), a body-centered cubic (bcc), or a simple cubic grid. The purpose of this is to not have any atoms too close to each other, since we usually have a strong repulsive force when atoms get close together, which would give very big forces. We also want to start with the atoms in a state from which we are able to quickly get to the state we want to study. If we for example want to study a liquid argon system, it is wise to start in an unstable crystal state, by for example using a low density or high temperature, so that the system would melt spontaneously when we start the simulation.

1.2 Calculation of forces

The forces are calculated from the derivatives of interatomic potentials, that usually only depend on the positions of the atoms. The potentials are generally of the form

$$U(\boldsymbol{r}) = \sum_{i<j} U_{ij}(r_{ij}) + \sum_{i<j<k} U_{ijk}(\boldsymbol{r}_i, \boldsymbol{r}_i, \boldsymbol{r}_k) + \ldots,$$

where $\boldsymbol{r}_i$ is the position of atom i, r_{ij} is the distance between atom i and j, U_{ij} is a two-particle potential depending only on the distance between two atoms, and U_{ijk} is a three-particle potential that usually also depends on the angle between three atoms. Higher-order contributions to the forces are also sometimes used, but these are very demanding to evaluate.

The potentials are often developed from quantum mechanical calculations, and when doing this one has to weigh the benefits of having a complex potential that models the interactions accurately, against having a less complex potential that will be easier to implement, and faster to evaluate. The limiting factor in any molecular dynamics calculation is the cost of doing simulations on high-performance computing clusters (like Abel at UiO), but luckily it seems like the progress in computational processing power still seems to almost follow Moore's Law[24], which states that the number of transistors on integrated circuits double approximately every two years[30], effectively halving the cost of doing a computation every two years.

In this example we will be using a potential first seen as early as 1924[21] called the Lennard-Jones potential after its creator, who used to to study the noble gas Argon. The potential is a two-particle potential with the following form

$$U(r_{ij}) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] = \varepsilon \left[\left(\frac{r_m}{r}\right)^{12} - 2\left(\frac{r_m}{r}\right)^{6} \right], \tag{1.1}$$

where σ is the distance between where the potential is zero (the equilibrium distance between the atoms), ε is related to the strength of the potential (the minimum value of the potential), and $r_m = 2^{1/6}\sigma$ is the interatomic distance where the potential is at its minimum. The r^{-12}-term is a repulsive term that describes overlap of electron orbitals (Pauli repulsion) and the r^6-term is an attractive term that describes dipole-dipole interactions (van der Waals forces).

Even though the potential is simple, it describes many properties of noble gases like Argon well, and its simplicity also means that the cost of calculating the forces between atoms is low. For these reasons it has been used in a lot of studies.

See fig. 1.1 for a plot of the potential using the parameters usually used for simulating Argon[13], $\sigma = 3.405$ Å and $\varepsilon = 0.010318$ eV.

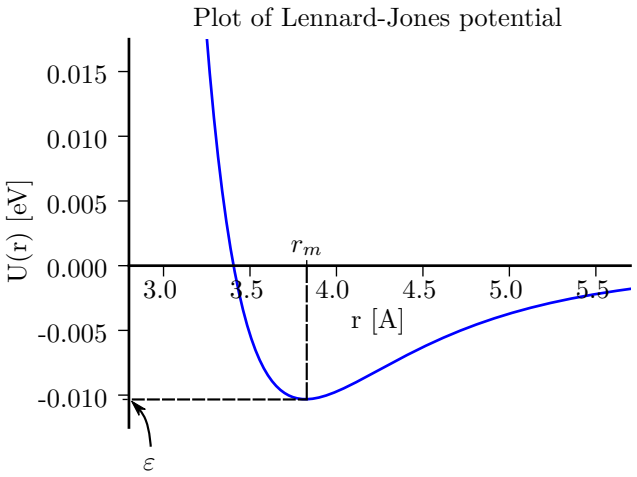


Figure 1.1: Plot of the Lennard-Jones potential, as stated in eq. (1.1). Using the parameters usually used for simulating Argon, $\sigma = 3.405$ Å and $\varepsilon = 0.010318$ eV[13].

1.2.1 Newton's third law

When evaluating two-particle forces like the Lennard-Jones potential there is a simple optimization that lets us halve the number of computations, by utilizing Newton's third law. We see that when evaluating $U(r_{ij})$, the force will have the same magnitude if we switch particle i and j. This means that when we have calculated the force $\boldsymbol{F}_{ij}$, from particle j on particle i, we know that the force on atom j from particle i will have the same magnitude, and we can simply add the opposite force to atom j, $\boldsymbol{F}_{ji} = -\boldsymbol{F}_{ij}$. This way we can skip half the force calculations, and only have to calculate the forces between particle i and particles $j > i$ in the main force loop.

See listings 1.2 and 1.3 for an example of how to implement force calculation using the Lennard-Jones potential, using this optimization.

If we are using higher-order potentials we can use the same optimization for the two-particle terms of the potential, but it is a bit more complicated for the terms depending on the positions of three or more particles, and the angles between them, but there are still similar optimizations that can be done if we are smart.

```
void calculateForces(System &system) {
    const vector<Atom*> &atoms = system.atoms();
    for (auto atom1 = atoms.begin(); atom1 != atoms.end(); ++atom1) {

        // Use Newton's third law to skip half the force calculations
        for (auto atom2 = atom1.next(); atom2 != atoms.end(); ++atom2) {
            vec3 force = calculateTwoParticleForce(*atom1, *atom2);

            (*atom1)->force() += force;
            (*atom2)->force() -= force; // Newton's third law
        }
    }
}
```

Listing 1.2: Implementation of `calculateForces` from listing 1.1. See listing 1.3 for example implementation of `calculateTwoParticleForce`.

```
vec3 calculateTwoParticleForce(Atom *atom1, Atom *atom2) {
    vec3 drVec = atom1->position() - atom2->position();

    double dr2 = drVec.lengthSquared();
    double dr6 = dr2*dr2*dr2;

    double LJforce = 24.0*(2.0 - dr6)/(dr6*dr6*dr2);
    vec3 force = drVec*LJforce;

    return force;
}
```

Listing 1.3: Implementation of `calculateTwoParticleForce` from listing 1.2, using the Lennard-Jones potential.

1.3 Integration scheme

To integrate Newton's equations of motion (time derivative denoted by a dot)

$$\begin{aligned}\dot{\boldsymbol{r}} &= \boldsymbol{v} \\ \boldsymbol{F} &= -\nabla U(\boldsymbol{r}(t)) = -\nabla U(t)\end{aligned}$$

for the intermolecular potential $U(\boldsymbol{r})$ there are a lot of different methods to choose between, ranging from the simple forward Euler method first described by Leonard Euler in 1768, to higher order "predictor-corrector" methods. The integrator we use need to fullfil certain demands. Since the equations of motion are reversible, so should our integration method be. We also want to be able to use as large timesteps as possible, to save computation time. Lastly, and perhaps most importantly, we want the energy of our system to be conserved, so the long-term drift in energy should be small. We only need to be able to make

statistically correct predictions about the trajectories of the atoms, so predicting the true trajectories of the atoms is not a priority as long as they are statistically correct.

It turns out that a deceptively simple method first described by Loup Verlet in 1967[45] often satisfies our needs in an integrator, being both very accurate over long simulation times, having a accumulated error in energy of the order $\mathcal{O}(\Delta t^2)$ (as shown in appendix A.2.3), and is time-reversible. This method also has the advantage that it is numerically cheap, compared to higher-order methods. The integration method is called Verlet integration.

1.3.1 Regular Verlet integration

The Verlet method has many variations, but the simplest form (the one used by Verlet in [45]) has the form

$$\boldsymbol{r}(t + \Delta t) \approx 2\boldsymbol{r}(t) - \boldsymbol{r}(t - \Delta t) + \boldsymbol{a}(t)\Delta t^2, \tag{1.2}$$

where Δt is the timestep, and $\boldsymbol{a}(t)$ is the velocity at time t. This form of the scheme has a truncation error in the position for one timestep of the order $\mathcal{O}(\Delta t^4)$ (see appendix A.1).

We see that the velocity is not explicitly calculated or used in this form of the sceme, but if we need it for our experiments we can estimate the velocity using a Taylor expansion around $\boldsymbol{r}(t \pm \Delta t)$, which gives

$$\boldsymbol{v}(t) = \frac{\boldsymbol{r}(t + \Delta t) - \boldsymbol{r}(t - \Delta t)}{2\Delta t},$$

which has a truncation error for one timestep of the order $\mathcal{O}(\Delta t^2)$ (see appendix A.1).

The implementation of the Verlet scheme is mostly straightforward, the only thing we have to take care of is what happens in the first timestep. When calculating the positions in the first step, $\boldsymbol{r}(0+\Delta t)$, we see from eq. (1.2) that we need the positions from the previous step, $\boldsymbol{r}(0-\Delta t)$. These positions are usually approximated using the initial velocity, as follows

$$\boldsymbol{r}(0 - \Delta t) = \boldsymbol{r}(0) - \boldsymbol{v}(0)\Delta t.$$

See listing 1.4 for an example of how to implement the Verlet integration scheme.

```
void integrateEquationsOfMotion(System &system, double dt) {
    for (Atom *atom : system.atoms()) {
        vec3 newPosition = 2.0*atom->position() - atom->oldPosition()
                         + atom->force()*dt*dt;
        atom->oldPosition() = atom->position();
        atom->position() = newPosition;
        atom->velocity() = (atom->position() - atom->oldPosition())
                         /(2.0*dt);
    }
}
```

Listing 1.4: Implentation of `integrateEquationsOfMotion` from listing 1.1, using regular Verlet integration.

1.3.2 Velocity Verlet

The most used form of the Velocity integration scheme is called the velocity Verlet method[41], and it has the form

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2}, \tag{1.3}$$

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \big[\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t)\big]\frac{\Delta t}{2}, \tag{1.4}$$

with the truncation error for one timestep Δt being of the order $\mathcal{O}(\Delta t^3)$ for both the position and the velocity, and the accumulated error in energy being of the order $\mathcal{O}(\Delta t^2)$ (see appendices A.1 and A.2 for more information). It can be shown that this form is equivalent to the regular Verlet method.

One advantage of this form compared to the regular Verlet method, is that it is self-starting. In the regular Verlet algorithm we need $\boldsymbol{r}(t-\Delta t)$ to compute $\boldsymbol{r}(t+\Delta t)$, which we do not have at $t=0$. This means that we have to approximate $\boldsymbol{r}(-\Delta t)$ somehow. In the velocity form of the algorithm we only need the positions, velocities and forces at time t to calculate $\boldsymbol{r}(t+\Delta)$.

The velocity Verlet algorithm is usually rewritten in the following way to optimize implementation on a computer. The new velocities $\boldsymbol{v}(t+\Delta t)$ can be written as

$$\boldsymbol{v}(t+\Delta t) = \tilde{\boldsymbol{v}}(t+\tfrac{1}{2}\Delta t) + \boldsymbol{a}(t+\Delta t)\frac{\Delta t}{2}, \tag{1.5}$$

where

$$\tilde{\boldsymbol{v}}(t+\tfrac{1}{2}\Delta t) = \boldsymbol{v}(t) + \boldsymbol{a}(t)\frac{\Delta t}{2}. \tag{1.6}$$

We see that eq. (1.6) can be used in updating the positions, so we rewrite eq. (1.3) to

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \tilde{\boldsymbol{v}}(t+\tfrac{1}{2}\Delta t)\Delta t. \tag{1.7}$$

Which leads us to the usual way of implementing the algorithm[3]:

- Calculate the velocities at $t + \frac{1}{2}\Delta t$ using eq. (1.6) (repeated here)

$$\tilde{\boldsymbol{v}}(t+\tfrac{1}{2}\Delta t) = \boldsymbol{v}(t) + \frac{\boldsymbol{F}(t)}{m}\frac{\Delta t}{2}.$$

- Calculate the new positions at $t + \Delta t$ using eq. (1.7) (repeated here)

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \tilde{\boldsymbol{v}}(t+\tfrac{1}{2}\Delta t)\Delta t.$$

- Calculate the new forces $\boldsymbol{F}(t + \Delta t)$.
- Calculate the new velocities at $t + \Delta t$ using eq. (1.5) (repeated here)

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t+\tfrac{1}{2}\Delta t) + \frac{\boldsymbol{F}(t+\Delta t)}{m}\frac{\Delta t}{2}.$$

This implementation minimizes the memory needs, as we only need to store one copy of $\boldsymbol{r}$, $\boldsymbol{v}$ and $\boldsymbol{F}$ at all times, compared to implementing eqs. (1.3) and (1.4) which needs to store the values of both $\boldsymbol{F}(t)$ and $\boldsymbol{F}(t + \Delta)$ to calculate the new velocities. Memory is usually abundant these days though, so this is not really an issue. It can also be shown that this implementation leads to less floating point truncations[13].

1.4 Boundary conditions

In theory we now have a working molecular dynamics program by combining listings 1.1 to 1.4. But if we start our simulations we will quicly see that the particles will start spreading out into space, since we have not implemented any kind of boundary conditions. The particles that are on the surface of our initial system will feel very different forces than the ones in the center, and will most likely not behave as intended. To remedy this we usually apply periodic boundary conditions. This means that we repeat the simulation box at the boundaries of the system, so that the atoms near the boundaries feel forces from atoms on the opposite side of the system, and in an uniform system all atoms will have a bulk-like environment.

Atom n has what we call an *image* in all other neighboring simulation boxes, created by the periodic boundaries. The position of the images of atom n can be calculated using

$$\boldsymbol{r}_n^{ijk} = \boldsymbol{r}_l + (iL_x, jL_y, kL_z), \tag{1.8}$$

where (L_x, L_y, L_z) is the dimensions of the simulation box, and (i, j, k) is the index of the neighbor box. We usually choose box $(0, 0, 0)$ as our "origin" box, with the corner of the box in the point $(0, 0, 0)$. In reality we still have the same number of atoms, but the atoms on near the boundaries of the simulation box will now feel forces from the atoms on the opposite side of the box, from the images of the atoms on the opposite side of the box.

The first thing we have to do to implement periodic boundary conditions is to check if any atoms have moved outside the boundaries of the system after each timestep, after updating the positions of the atoms. If they have moved outside the boundaries of the system we see from eq. (1.8) that we can translate them back into the system by adding or subtracting an appropriate number of system sizes L_i from the coordinates that are outside the box. If the boundaries of our system are $x_i \in [0, L_i]$, with $\boldsymbol{r} = (x_1, x_2, x_3)$, we can translate atomic positions *outside* the boundaries to the correct positions *inside* the boundaries by using the modulo operator. By finding the remainder of dividing the coordinates of an atom with the system size, we get back the position of the atom translated back inside the boundaries, as follows

$$x_i^{000} = x_i^{ijk} \bmod L_i.$$

When implementing this we have to be wary of what happens if we have negative coordinates, as the modulo with negative number has different implementations in different programming languages. To avoid this problem we usually just add one system size to each coordinate before using the modulo operator, to ensure that the coordinates are positive. In doing this we assume that no atoms have moved more than one system size in negative direction, but if the timestep in the simulations is set correctly, atoms should never move as far as one system size in any direction in just one timestep.

1.4.1 Minimum image convention

A consequence of using periodic boundary conditions is that each atom now feels the force from an infinite number of atoms. To avoid having to do an infinite number of evaluations of the potential we implement something called the *minimum image convention*. This implies that we only calculate the force between atom n and the *nearest* image of each atom m, effectively limiting the potential

to half the size of the system in each direction. When doing this truncation and simulating "bulk" or "infinte" systems, we do an approximation that might have some consequences in some cases, but this is rarely a problem. See [3, Section 1.5] for a discussion on this matter.

To find the distance between atom n and the closest image of atom m we can calculate the distance between n and any image of m, $\boldsymbol{r}_{nm}$, and then check if any of the components of this vector is larger than half the system size in that direction. If a component is larger than half the system size, we subtract (a whole) system size to get the correct distance. See listing 1.5 for an example of a function that finds the distance between a point u and the closest image of a point v using the minimum image convention.

```
double calculateDistanceSquaredUsingMinimumImageConvention(
    const vec3 &u, const vec3 &v,
    const vec3 &systemSize, const vec3 &halfSystemSize) {

    vec3 dr = u - v;
    for (int dim = 0; dim < 3; dim++) {
        if (dr[dim] >= halfSystemSize[dim]) dr -= systemSize[dim];
        else if (dr[dim] < -halfSystemSize[dim]) dr += systemSize[dim];
    }
    return dr.lengthSquared(); // Avoid calculating √(dr²), return dr² instead
}
```

Listing 1.5: An example of how to find the distance between two points `u` and `v` in a periodic system of size `systemSize` using the *minimum image convention*. We calculate the distance squared to avoid taking the square root, since this is a slow operation.

1.5 Optimization via force truncation

If we try to simulate systems with a lot of atoms using the program we now have developed, we see that the number of evaluations of the potential quickly grow with the number of atoms, scaling as $\mathcal{O}(N^2)$. To optimize the program we can limit the number of evaluations by realizing that the Lennard-Jones potential decays as r^{-12}, meaning that the force between most atoms will be neglible (see fig. 1.1 for a plot of the potential).

Using the parameters for Argon, we find from eq. (1.1) that the equilibrium distance of the potential is $r_{\text{eq}} = 2^{1/6}\sigma \approx 3.8$ Å. We also find that the value of the potential has decreased to 21% of the equilibrium value at a distance $r_{ij} = 5.5$ Å, and to 0.5% of the equilibrium value at a distance $r_{ij} = 3\sigma \approx 10.2$ Å. From this we decide to truncate the potential at a cutoff distance $r_{\text{cutoff}} = 3\sigma$.

The naive implementation of this cutoff length is to do a test inside the force calculation (for example in `calculateForces` in listing 1.2) and see if the distance between atom i and j is greater than the cutoff distance, $r_{ij} > r_{\text{cutoff}}$. This approach this still requires the calculation of a lot of interatomic distances, so what we do instead is to implement so-called *cell-lists*.

1.5.1 Cell lists

To truncate the force using cell-lists we divide the system into (3d) cells (or boxes) of size

$$l = L/n,$$

where n is the number of cells in a direction. We can calculate the number of cells from the cutoff length r_{cutoff} using the floor function $\lfloor x \rfloor$ as follows

$$n = \lfloor L/r_{\text{cutoff}} \rfloor.$$

Using the floor function guarantees that $l \geq r_{\text{cutoff}}$.

The truncation is now done by only calculating the force between atom i and all atoms in the cell it belongs to, and between it all atoms in the 26 neighboring cells. Since $l \geq r_{\text{cutoff}}$, this means that we include all atoms within a distance of at least r_{cutoff} in the force calculations. See fig. 1.2 for a 2-dimensional illustration of this.

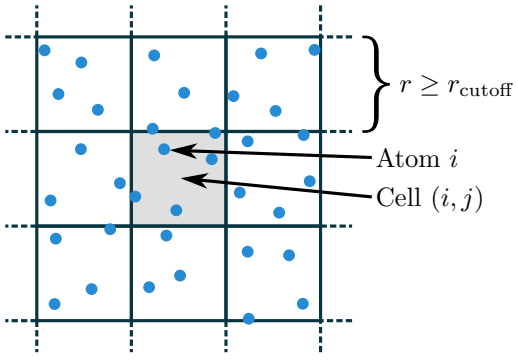


Figure 1.2: An illustration of cell lists in 2 dimensions. We truncate the potential at r_{cutoff} by only calculating the force between atom i and all atoms in the cell of that atom, and between that atom and all atoms in the 8 neighbor cells (26 neighbor cells in 3 dimensions).

One issue that arises when using the cell lists is how to utilize Newton's third law. We see that after calculating the force between all atoms in cell (i, j, k) and all

atoms in the 26 neighboring cells, while adding the calculated forces to the other atoms using Newton's third law, we must not include that cell (cell (i, j, k)) in any other force calculations this timestep, or else we will get double contributions from atoms in that cell. One simple way of solving this is to keep a list of all cells we have calculated the forces on (and from) so far, and then check against that list every time we are calculating the force from a neighbor cell. But if we loop through the cells in the same order every time, we see that it would perhaps be more efficent to make a list over neighbor cells for each cell, so that we can just loop through a list of neighbors. The time spent on this in the simulations is nevertheless neglible compared to the evaluations of the potential.

An implementation of the calculation of forces using cell lists can be seen in listings 1.6 to 1.8. In these examples we do not use Newton's third law for optimization, to shorten the code and make the example simpler.

```
void calculateForces(System &system) {
    ivec3 nCells;
    vector<Atom*> cells =
        sortAtomsIntoCells(system, cutoffLength, nCells);

    // Loop over all cells
    for (int i = 0; i < nCells[0]; i ++)
    for (int j = 0; j < nCells[1]; j ++)
    for (int k = 0; k < nCells[2]; k ++)
    {{{
        for (Atom *atom : cells[i][j][k]) {
            atom->force() +=
                calculateForceFromNeighborCells(
                    cells, nCells, atom, i, j, k
                );
        }
    }}}
}
```

Listing 1.6: An example of an implementation of the force calculation `calculateForces` from listing 1.1, using the Lennard-Jones potential with a cutoff length for the force, and cell lists. Notice that we do not use Newton's third law, to simplify the example.

When implementing cell-lists we see that we need to have accurate lists of the atoms in each cell. But if we realize that the atoms rarely move very far in one timestep, we see that we can get away with only updating the cell-lists every other timestep. How often we need update the lists depends on the temperature (the average velocity of the atoms) and the timestep.

```
vector<Atom*> sortAtomsIntoCells(
    System &system, double cutoffLength, ivec3 &nCells) {

    nCells = floor(system.size() / cutoffLength);
    ivec3 boxSize = system.size() / vec3(nCells);

    vector<Atom*> cells;
    for (Atom *atom : system.atoms()) {
        ivec3 index = floor(atom->position() / boxSize);
        cells[index[0]][index[1]][index[2]].push_back(atom);
    }
    return cells;
}
```

Listing 1.7: An example of an implementation of `sortAtomsIntoCells` from listing 1.6. This listing shows how to sort atoms into cells for the cell list optimization described in section 1.5.

```
vec3 calculateForceFromNeighborCells(
    vector<Atom*> &cells, ivec3 &nCells, Atom *atom1,
    int i1, int j1, int k1) {

    vec3 force = zeros<vec3>();
    // Loop over 27 neighbor cells (including self)
    for (int di = -1; di <= 1; di++)
    for (int dj = -1; dj <= 1; dj++)
    for (int dk = -1; dk <= 1; dk++)
    {{{
        // Periodic boundary conditions
        int i2 = (i1 + di + nCells[0]) % nCells[0];
        int j2 = (j1 + dj + nCells[1]) % nCells[1];
        int k2 = (k1 + dk + nCells[2]) % nCells[2];

        // Loop over atoms in neighbor cell
        for (Atom *atom2 : cells[i2][j2][k2]) {
            if (atom1 == atom2) continue; // Skip i == j
            force() += calculateForceBetweenTwoAtoms(atom1, atom2);
        }
    }}}
    return force;
}
```

Listing 1.8: An example of an implementation of `calculateForceFromNeighborCells` from listing 1.6. This listing shows how to calculate the force on an atom (`atom1`), from the atoms in the cell it belongs to (`cells[i1][j1][k1]`), and from the atoms in all 26 neighbor cells.

1.6 Observables

We are now ready to start doing some simple simulations, and observe some properties of the system. One of the most basic statistics we can observe we is the temperature and pressure in a system, and we use results from statistical mechanics to justify that we can simply take the average over all atoms to calculate these observables.

An example of a function that samples statistics for the observables can be seen in listing 1.9 (note that `diffusionSample` will be defined later, in listing 7.5).

```
void sample(System &system) {
    double temperature = temperatureSample(system);
    double pressure = pressureSample(system, temperature);
    double rSquared = diffusionSample(system);
}
```

Listing 1.9: Implementation of the function `sample` from listing 1.1. See listing 1.10, listing 1.11, and listing 7.5 for example implementation of the functions used.

1.6.1 Temperature

According to the equipartition principle the average total kinetic energy, for a system consisting of N particles with three degrees of freedom each, can be related to the temperature of the system via

$$\langle E_k \rangle = \frac{3}{2} N k_{\mathrm{B}} T,$$

where T is the temperature of the system, and k_B is the Boltzmann constant. We calculate the average kinetic energy of a system using

$$\langle E_k \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{2} m_i v_i^2,$$

where m_i and $v_i = |\boldsymbol{v}_i|$ is respectively the mass and speed of atom i. From this we find the temperature of the system as

$$T = \frac{2}{3} \frac{\langle E_k \rangle}{N k_B} = \frac{1}{3N^2 k_B} \sum_{i=1}^{N} m_i v_i^2.$$

See listing 1.10 for an example of how to calculate the temperature in a molecular dynamics simulation.

```
double temperatureSample(System &system) {
    double kineticEnergy = 0.0;
    for (Atom *atom : system.atoms()) {
        kineticEnergy += atom->velocity().lenghtSquared();
    }

    kineticEnergy *= 0.5;
    double temperature = 2.0*kineticEnergy/(3.0*system.nAtoms()
                                            *boltzmannConstant);  // SI units
    return temperature;
}
```

Listing 1.10: An example of how to calculate the temperature in a molecular dynamics simulation. Example implementation of `temperatureSample` from listing 1.9.

1.6.2 Pressure

To measure the pressure when using potentials with pairwise additive interactions, like the case is for our example program with the Lennard-Jones potential, we can use a method derived from the virial equation for the pressure[13, Section 4.4]. In a volume V with particle density $\rho = N/V$, the average pressure is

$$P = pk_{\mathrm{B}}T + \frac{1}{dV}\left\langle \sum_{i<j} \boldsymbol{F}(\boldsymbol{r}_{ij}) \cdot \boldsymbol{r}_{ij} \right\rangle, \tag{1.9}$$

where $\boldsymbol{F}(\boldsymbol{r}_{ij})$ is the force between particle i and j, and $\boldsymbol{r}_{ij}$ is the distance between the particles. Note that this expression for the pressure has been derived for a system at constant N, V and T, whereas our simulations are performed at a constant N, V, and energy E.

We see that we need the force from each atom j on atom i, $\boldsymbol{F}(\boldsymbol{r}_{ij})$ to calculate the pressure, so for efficiency we should calculate the contribution to the pressure, $\boldsymbol{F}(\boldsymbol{r}_{ij}) \cdot \boldsymbol{r}_{ij}$, while doing the force calculations. The contribution to the pressure should then be stored so we can calculate the average in eq. (1.9) later.

An example of how to calculate the pressure in a molecular dynamics similation can be seen in listing 1.11.

```
double pressureSample(System &system, double temperature) {
    double pressure;
    for (Atom *atom : system.atoms()) {
        pressure += atom->pressure();
    }
    pressure /= (3.0*system.volume());
    double density = system.nAtoms()/system.volume(); // Assume homogeneous
    pressure += density*boltzmannConstant*temperature; // SI units
    return pressure;
}
```

Listing 1.11: An example of how to calculate the pressure in a molecular dynamics simulation. Example implementation of `pressureSample` from listing 1.9. Note that this function needs the temperature of the system as input, and assumes that the system is homogeneous, so we can estimate the density using $\rho = N/V$. We assume that the contribution to the pressure from each atom $\sum_{i<j} \boldsymbol{F}(\boldsymbol{r}_{ij}) \cdot \boldsymbol{r}_{ij}$ (stored as `atom->pressure()`) has been calculated previously. This is usually calculated while calculating the forces between the atoms, since we need $\boldsymbol{F}(\boldsymbol{r}_{ij})$. See section 1.6.2 for more information.

Chapter 2

Ensembles

We have now developed a molecular dynamics program that simulates simple systems like Argon, with constant number of particles N, constant volume V and constant energy E, which means that the system can not exchange particles or energy with the environment. This forms a statistical ensemble called the microcanonical ensemble or the NVE-ensemble, which has some implications for what we can simulate and measure. Other ensembles that might be of interest is the canonical ensemble (constant N, V and temperature T) and the grand canonical ensemble (constant N, pressure P, and temperature T).

The most often studied ensemble other than the microcanonical is the canonical, with constant temperature instead of energy. To study this we need a way of controlling the temperature of the system, which is usually done by using a *thermostat*. A thermostat simulates the system being in contact with a heat bath with temperature T_{bath}. The temperature is defined via the kinetic energy of the system, so we know we have to somehow control and modify the velocities of the atoms in the system to control the temperature.

2.1 Berendsen thermostat

Perhaps the simplest example of a thermostat is the Berendsen thermostat[4], which rescales all velocities by multiplying them with a factor γ

$$\gamma = \sqrt{1 + \frac{\Delta t}{\tau}\left(\frac{T_{\text{bath}}}{T} - 1\right)},$$

where Δt is the timestep used in the simulations, τ controls the strength of the thermostat, T is the temperature of the system and T_{bath} is the temperature

of the simulated heat bath. Setting $\tau = \Delta t$ makes the thermostat change the temperature of the system so it is exactly equal to T_{bath}. The velocities can either be multiplied by this factor every timestep, or every n-th timestep. An example of how to apply the Berendsen thermostat can be seen in listing 2.1.

```
void applyBerendsenThermostat(System &system, double T, double Tbath,
    double dt, double tau) {

    double gamma = sqrt(1 + dt/tau(Tbath/T - 1));
    for (Atom *atom : system.atoms())
        atom->velocity() *= gamma;
    }
}
```

Listing 2.1: Example of how to implement the Berendsen thermostat.

The Berendsen thermostat is very good at controlling and changing the temperature of a system, but it does not sample the canonical ensemble very well. This is because we change the velocity of *all* atoms at every n-th timestep, which is not physically realistic. This means that we should not use this thermostat when trying to sample the canonical ensemble, but we often use it to heat up or cool down a system, to reach a wanted temperature.

Many thermostats similar to the Berendsen thermostat exist but they all suffer from the fact that they scale the velocity of all particles, giving unphysical behaviour.

2.2 Andersen thermostat

The Andersen thermostat is a more physically realistic thermostat, which simulates hard collisions between atoms in the system and atoms in the heat bath. We do not actually simulate any extra particles, but we we assign new random velocities to a random fraction of the atoms, the fraction and magnitude of the velocity determined by the strength of the thermostat and the temperature of the thermostat.

This thermostat uses the following procedure

- For each atom generate a uniform random number u in the interval $[0, 1]$.
- If this random number is less than $\Delta t/\tau$,

$$u < \frac{\Delta t}{\tau},$$

we assign the atom a new, normally distributed velocity with standard deviation

$$\sigma_v = \sqrt{\frac{k_B T_{\text{bath}}}{m}}.$$

In this thermostat τ can be seen as a collision time, and τ should have about the same value as in the Berendsen thermostat.

The Andersen thermostat samples the canonical ensemble well, but disturbs the dynamics of for example lattice vibrations. We should avoid using this thermostat when measuring properties directly connected to the movement of each particle, for example diffusion, since we so abruptly change the velocity and trajectory of the particles.

2.3 Nosé-Hoover thermostat and Nosé-Hoover chains

The Nosé-Hoover thermostat is an advanced thermostat that generates a correct canonical ensemble and give very accurate dynamics[13, section 6.1]. The effect of applying Nosé-Hoover type thermostats it that we get an additional *friction*-term to the forces on the atoms, instead of directly changing the velocities as with the Andersen and Berendsen thermostats.

2.3.1 Nosé-Hoover thermostat

The simplest form of the Nosé-Hoover type thermostats is the Nosé-Hoover thermostat. This introduces a thermodynamic friction coefficient ξ in the equations of motion, and we get the following equations of motion (the time derivative noted by a dot)

$$\dot{\boldsymbol{r}} = \boldsymbol{v} \tag{2.1}$$

$$\boldsymbol{F} = -\nabla U(\boldsymbol{r}) - \xi m \boldsymbol{v}, \tag{2.2}$$

where the friction coefficient generally depends on time, $\xi = \xi(t)$, and the choice of $\xi(t)$ determines the characteristics of the thermostat. See appendix B for a derivation of this.

If we try to integrate these equations of motion using our integrator of choice, the velocity Verlet algorithm, a problem with using this thermostat will become

apparent. The velocity Verlet integrator has the followin form (from eqs. (1.3) and (1.4))

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2}$$
$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \big[\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t)\big]\frac{\Delta t}{2}.$$

If we insert eq. (2.2) into these equations (using $\boldsymbol{F} = m\boldsymbol{a}$) we get

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \left[-\frac{\nabla U(t)}{m} - \xi(t)\boldsymbol{v}(t)\right]\frac{\Delta t^2}{2}$$
$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \Bigg[-\frac{\nabla U(t)}{m} - \xi(t)\boldsymbol{v}(t) - \frac{\nabla U(t+\Delta t)}{m} - \xi(t+\Delta t)\boldsymbol{v}(t+\Delta t)\Bigg]\frac{\Delta t}{2},$$

where $U(\boldsymbol{r}(t)) = U(t)$. The problem is now apparent: to calcuate $\boldsymbol{v}(t+\Delta t)$ we need to already know $\boldsymbol{v}(t+\Delta)$, which turns the integrator into a *implicit* integrator. For this reason the Nosé-Hoover thermostat can be implemented using a predictor-corrector scheme, or solved iteratively[13, Appendix E.2]. This has the disadvantage that the solution is no longer time reversible. A solution to this has been proposed by Martyna et al. in [29], where they develop a set of explicit reversible integrators using the Louiville approach for this type of extended systems, similar to the way we derive the velocity Verlet algorithm in appendix A.2.

Choice of ξ

In the so-called Nosé-Hoover thermostat the choice of ξ is

$$\xi = \frac{sp_s}{Q}, \tag{2.3}$$

and the time-evolution of ξ is described by

$$\dot{\xi} = \left(\sum_{i=1}^{N}\frac{p_i^2}{m_i} - 3Nk_BT\right)/Q. \tag{2.4}$$

Here s is a degree of freedom introduced to the Lagrangian and Hamiltonian of the system, p_s is the momentum associated with s, and Q the "mass" associated with s. Q controls the coupling to the heat bath, and has to be chosen with care. A large value for Q leads to a weak coupling.

An important result derived by Hoover in [17] is that this choice of ξ (eqs. (2.3) and (2.4)) is the *only* choice that can lead to a canonical distribution.

2.3.2 Nosé-Hoover chains

It can be shown that the Nosé-Hoover thermostat (eqs. (2.3) and (2.4)) only generates a correct canonical distribution for molecular systems in which there are no external forces, and the center of mass remains fixed [13, Appendix B.2.1][16]. The last condition can be obeyed if we initialize the system with a net zero center-of-mass velocity, which we usually do in our simulations to avoid drift. If we want to simulate systems with an external force, for example to introduce flow, or a system where the center off mass is not fixed, we can use what is called Nosé-Hoover *chains*, as proposed by Martyna et al.[28].

Nosé-Hoover chains is a scheme where we use a Nosé-Hoover thermostat which is coupled to another thermostat, or a whole chain of thermostats. In the Nosé-Hoover chains thermostat[28] we get the following equations of motion

$$\begin{aligned}\dot{\boldsymbol{r}} &= \boldsymbol{v} \\ \boldsymbol{F} &= -\nabla U(\boldsymbol{r}) - \frac{p_{\xi,1}}{Q_1} m\boldsymbol{v},\end{aligned}$$

which we see have a similar form to the equations we get when using a regular Nosé-Hoover thermostat (eqs. (2.1) and (2.2)), with a friction term $\boldsymbol{v}p_{\xi,1}/Q_1$ added to the force on the atoms. The equations of motion for the thermostats, with M coupled thermostats, are

$$\begin{aligned}\dot{\xi}_k &= \frac{p_{\xi,k}}{Q_k} \qquad \text{for } k = 1, \ldots, M \\ \dot{p_{\xi,1}} &= \left(\sum_i \frac{p_i^2}{m_i} - 3Nk_BT \right) - \frac{p_{\xi,2}}{Q_2} p_{\xi,1} \\ \dot{p_{\xi,k}} &= \left[\frac{p_{\xi_k - 1}^2}{Q_{k-1}} - k_BT \right] - \frac{p_{\xi,k+1}}{Q_{k+1}} p_{\xi,k} \\ \dot{p_{\xi,M}} &= \left[\frac{p_{\xi_M - 1}^2}{Q_{M-1}} - k_BT \right].\end{aligned}$$

As with the Nosé-Hoover thermostat, the regular velocity Verlet integrator can not be used with this thermostat, but a derivation of an integrator for these equations of motion can be seen in [13, Appendix E.2.1] and [29], where they again use the Liouville approach similar to the one we used to derive the velocity Verlet integrator in appendix A.2.

The Nosé-Hoover chains thermostat will generate a canonical distribution even in a system with external forces, and center of mass that is not fixed[13, Appendix B.2.2], and will for example work well in a simulation where we introduce flow by some external force.

Chapter 3

Molecular dynamics program used for simulations

To study an advanced system like silica and water we need a more advanced potential than the Lennard-Jones potential. We know water likes to have a certain H-O-H angle, and that water molecules have strong hydrogen bonds between them that make water have some unique properties. Similarly we know that silica usually appears in tetrahedral structures, with certain Si-O-Si angles more stable than others. To simulate the forces that make structures like that appear we need to use something more advanced than a diatomic potential Lennard-Jones, so we use a higher order potential. We would also like to be able to simulate chemical reactions like transfer of oxygen atoms from silica to water, which requires further modifications of the simulations.

The potential implemented in the program we use in this thesis has been developed at the University of Southern California (USC) by P. Vashishta, Rajiv K. Kalia, José P. Rino and Ingvar Ebbsjö[44] (see also[38, 39]). The potential contains two-body and three-body terms, and allows oxygen atoms to be transferred between silica and water. See section 3.1 for more details on the potential.

The potential is implemented in a `FORTRAN 77` program, which has been parallellized and highly optimized for running on high-performance computing clusters like Abel. The program is unfortunately closed-source, and we are not allowed to distribute this openly.

The program implements both the fast and simple Berendsen thermostat and the more sophisticated Nosé-Hoover chains thermostat to simulate the NPT-ensemble, and a thermostat for the NPT ensemble. It also implements so-called variable time-step integrators, which saves CPU-time when we have silica and water in the same system. See section 3.2 for more information about this.

An article where the program has been used to study silica and water can be seen in[38], where they do simulations of systems with 1 billion atoms. See also the supplements to the article[39].

3.1 Potential

The interatomic potential[44] we use for for both silica and water consists of a two-body and a three-body part, and has the form

$$E_{\text{tot}} = \sum_{i<j} V_{ij}^{(2)}(r_{ij}) + \sum_{i<j<k} V_{ijk}^{(3)}(\boldsymbol{r}_{ij}, \boldsymbol{r}_{ij}),$$

for

$$1 \leq \{i, j, k\} \leq N.$$

$V_{ij}^{(2)}$ is the two-body term, which consists of four terms that take into account steric repulsion, charge-charge (Coulomb), charge-dipole, and dipole-dipole (van der Waals) interactions. The two-body term only depends on the interatomic distance between atom i and j, $|\boldsymbol{r}_{ij}| = r_{ij} = r$, and it has the form

$$V_{ij}^{(2)}(r) = \underbrace{\frac{H_{ij}}{r^{\eta_{ij}}}}_{\text{steric repulsion}} + \underbrace{\frac{Z_i Z_j}{r} e^{-r/r_{1s}}}_{\text{Coulomb}} - \underbrace{\frac{D_{ij}}{2r^4} e^{-r/r_{4s}}}_{\text{charge-dipole}} - \underbrace{\frac{w_{ij}}{r^6}}_{\text{van der Waals}},$$

where the parameters are

Steric repulsion

- H_{ij} controls the strength of the steric repulsion
- η_{ij} is the strength/exponent of the steric repulsion

Charge-charge (Coulomb) interaction

- Z_i is the charge associated with atom i
- r_{1s} is the screening length for the interaction

Charge-dipole interaction

- D_{ij} controls the strength of the interaction
- r_{4s} is the screening length for the interaction

Dipole-dipole (van der Waals) interaction

- w_{ij} controls the strength of the interaction
- r_{4s} is the screening length for the interaction

$V_{ijk}^{(3)}$ is the three-body term, which take into account bending and stretching of covalent bonds. This term depends on the distances between atom i, j and k, and also the angle θ_{ijk} between the atoms. The term has the form

$$V_{jik}^{(3)}(\boldsymbol{r}_{ij}, \boldsymbol{r}_{ik}) = B_{ijk} \underbrace{\exp\left(\frac{\xi}{r_{ij} - r_0} + \frac{\xi}{r_{ij} - r_0}\right)}_{\text{bond-stretching}} \underbrace{\frac{\left(\cos\theta_{ijk} - \cos\theta_0\right)^2}{1 + C_{ijk}\left(\cos\theta_{ijk} - \cos\theta_0\right)^2}}_{\text{bond-bending}},$$

for

$$\{r_{ij}, r_{ik}\} \leq r_0.$$

The parameters of the three-body term are as follows

- B_{ijk} controls the strength of the three-body interaction

Bond-stretching

- ξ controls the strength of the bond-stretching
- r_0 is the cutoff distance for the three-body interaction

Bond-bending

- C_{ijk} controls the strength of the bond-bending
- θ_{ijk} is the angle between $\boldsymbol{r}_{ij}$ and $\boldsymbol{r}_{ik}$
- θ_0 is the angle at which the three-body term vanishes

The parameters used in our simulations were chosen by first determing good parameters for pure water and silica independently, and then interpolating between these parameters to allow transfer of oxygen atoms between silica and water. See for example [44] for more details on this. The actual details of how oxygen is transferred between silica and water is based on the number of silicon and hydrogen neighbors an oxygen atom has, but the details of this method is outside the scope of this thesis.

The potential is further optimized by linearizing it, meaning that the potential is calculated for example for different distances r_{ij} and angles θ_{ijk} when the program starts, and then the forces can be looked up directly in a table without having to evaluate the potential, when we run the simulations.

3.2 Integrator

The program implements the Nosé-Hoover thermostat described in section 2.3.2 to sample the microcanonical ensemble, using a reversible multiple time-scale integrator derived using the Trotter factorization of the Liouville propagator[43], in a similar way to the way we derive the velocity Verlet algorithm in appendix A.2. The program also has a reversible integrator for NVT (canonical) and NPT (isothermal–isobaric) ensembles[29] derived in a similar way.

See fig. 3.1 for a plot of the energy over 100 000 timesteps. We see that the energy is very well conserved, with a relative increase in energy of just 2e-6 over 100 000 timesteps.

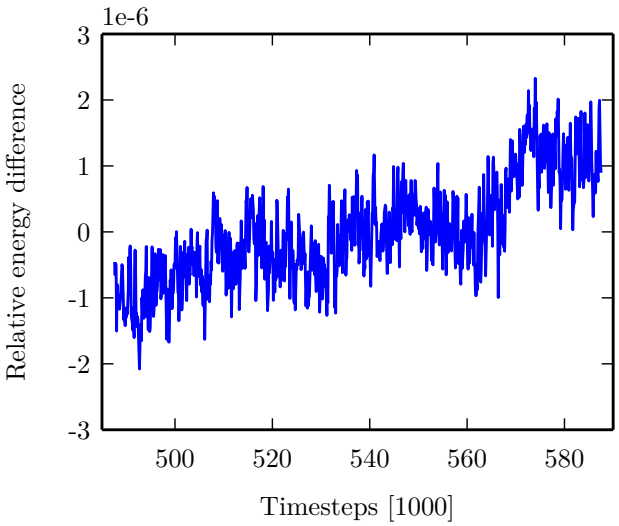


Figure 3.1: Plot of the relative energy change in a molecular dynamics simulation of water in nanoporous silica, with a total of approximately 400 000 atoms, 111k SiO_2 units and 19k H_2O-units. Simulations were done in the NVE-ensemble, and we have plotted 100 000 timesteps of 0.050 picoseconds.

The multiple time-scale integrator utilizes the fact that the vibrational frequencies in water is much higher than the frequencies of silica, which means that we can use larger timesteps to integrate the motions of the silica molecules than the water molecules. When using the program we use timesteps that are calculated from these vibrational frequencies, to make sure we have small enough timesteps. We used a main timestep of approximately 0.050 picoseconds in all simulations.

Part II

Fractures

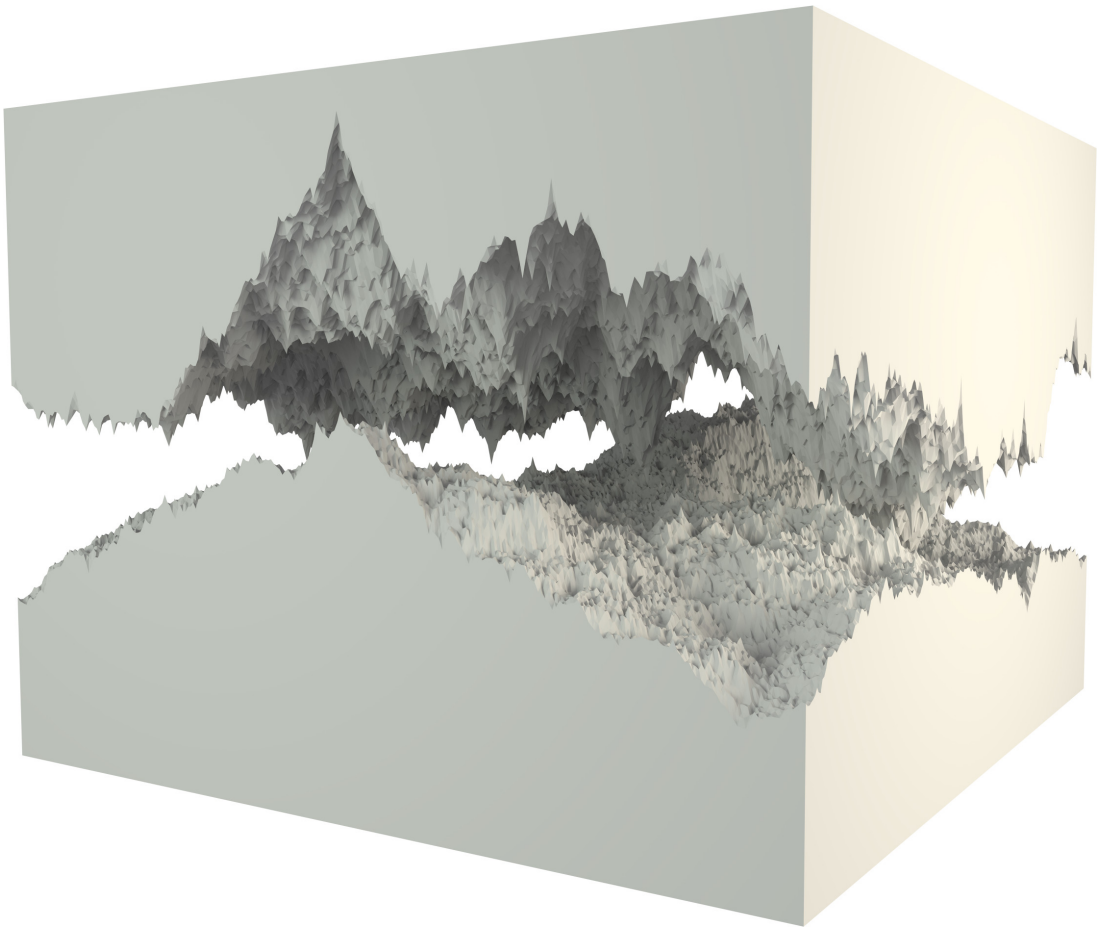


Figure 3.2: A randomly generated fracture.

Introduction

We want to study the behaviour of water trapped in nanoscale pores and fractures in silica, so need a way to generate and characterize such structures. We choose to model a fracture as two surfaces, with the volume between the surfaces as the void fraction. This makes it easier to make fractures, since we only need to create two surfaces to get a fracture.

To generate realistic surfaces we could have used scans of the structures we want to simulate, but the problem with this approach is that the resulting fracture will depend a lot on how we interpret the image, and that we can not easily generate a lot of samples of surfaces. To avoid this we use fractals to describe surfaces, and use this to randomly generate surfaces and fractures that are statistically similar to real fractures. Like a lot of phenomena in nature, fractures and surfaces can be very well described by the theory of fractals[25], so we think that this method should give good results.

What makes a *fractal* fractal, or what characterizes a fractal, does not have a rigorous definition, but in general a fractal is something that looks similar to itself at different length scales. A fractal might be *identical* to itself at different length-scales (self-similar), or be *statistically similar* to itself (statistically self-similar).

Chapter 4

Fractals and fractures

To generate a fractal surface we use fractional Brownian motion (fBm), introduced by Mandelbrot and van Ness in 1968[27]. Fractional Brownian motion is a generalization of Brownian motion, which is the random motion of particles suspended in a fluid, which comes from their collisions with the atoms and molecules in the fluid.

Fractional Brownian motion is a process that generates data that is fractal, in the sense that it is self-similar. The data generated by this process can be characterized by a parameter denoted H, often called the Hurst exponent. H is related to the autocorrelation of a data set, and is a number between 0 and 1. It has been shown that fractures and other phenomena in nature have a Hurst exponent of around 0.75, for over eleven decades of length scales[33], so we will try to generate fractures with this Hurst exponent for our simulations.

Samples of fBm with different Hurst parameters will differ in what can quantitatively be called the "roughness" or the "randomness" of the data, as can be seen in fig. 4.1, where we have plotted some samples of fBm with different Hurst exponents. We see that a low Hurst exponent leads leads to very rough or random data, and a high exponent to smoother data.

The Hurst exponent and the use of it as a means of characterizing a dataset was developed in the field of hydrology, as seen in [18, 19], where it was used to determine the optimal dam sizing for the Nile river's, by studying the large fluctuations in the flow rate of the river, which there are extensive records of. The exponent is denoted H in honor of both Harold Hurst, who was the lead researcher in these studies, and in honor of Otto Hölder.

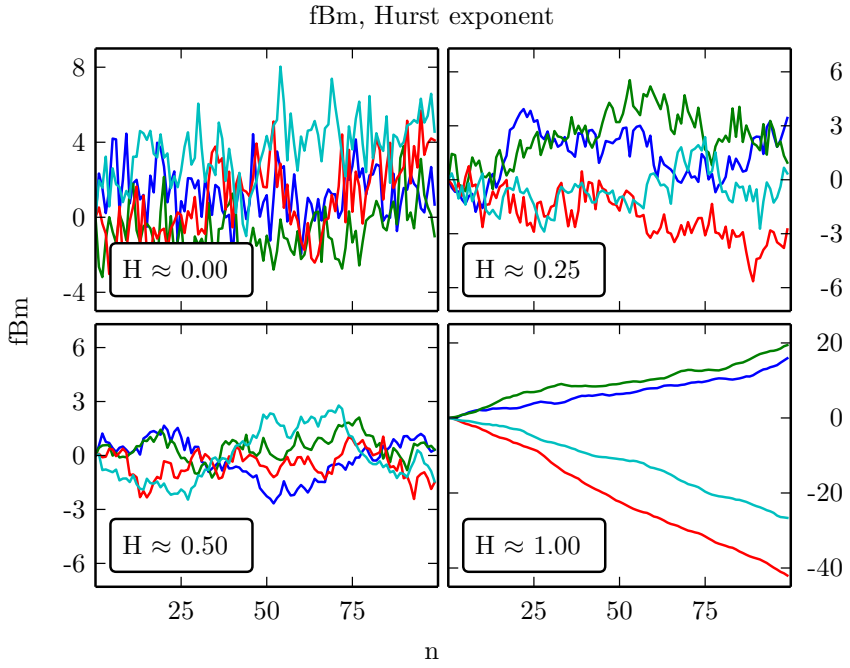


Figure 4.1: Samples of fractional Brownian motion (fBm) with different Hurst exponents, generated using the built-in Matlab function `wfbm`, which uses uses a wavelet-based synthesis method[1] for generating fBm.

4.1 Hurst exponent

One of the earliest methods for measuring the Hurst exponent is a statistical method developed by Hurst in his studies of the Nile river[19][18]. This method is called *rescaled range analysis* and was designed for use on 1-dimensional time series. The method has been generalized to higher dimensions[9], but the original 1-dimensional form is shown here.

4.1.1 Rescaled range analysis

We have a 1-dimensional time series $f(t)$. The time series is first divided into intervals of length τ. The average over each interval of length τ is

$$\langle f \rangle_\tau = \frac{1}{\tau} \sum_{t=1}^{\tau} f(t).$$

We let F be the accumulated deviation from the mean

$$F(t,\tau) = \sum_{t'=1}^{t} \big(f(t') - \langle f \rangle_\tau\big).$$

The difference between the maximum and minimum of the accumulated deviation from the mean is the *range* R

$$R(\tau) = \max_{1\le t\le\tau} \big(F(t,\tau)\big) - \min_{1\le t\le\tau} \big(F(t,\tau)\big).$$

The standard deviation S of the time series is estimated using

$$S^2 = \frac{1}{\tau}\sum_{t=1}^{\tau} \big(f(t) - \langle f \rangle_\tau\big)^2.$$

Hurst found that the observed *rescaled range*, R/S, for many time series is described by the empirical relation[10]

$$\frac{R}{S} = \left(\frac{\tau}{2}\right)^H \sim \tau^H.$$

We now see that we can estimate the Hurst exponent by a linear fit of the form

$$\log\left(\frac{R}{S}\right) \sim H \log\tau,$$

where we find H as the slope of the linear fit.

4.2 Detrending moving average

Estimating the Hurst exponent of a surface is not trivial, so to measure this we use a method called detrending moving average (DMA), developed for 1-dimensional data by E. Alessio, A. Carbone et al.[2], and later generalized to higher dimensions by A. Carbone [6].

After trying out some different methods for estimating the Hurst exponent, we ended up choosing this method both because it is easy to understand and implement, and because it has been shown to give good results, as we will also confirm later. A more detailed comparison of different methods for estimating the Hurst exponent can be seen in [37], where they find that DMA and DFA (detrended fluctuation analysis) overall perform better than FA (fluctuation analysis), also being less sensitive to the choice of scaling range.

4.2.1 Detrending moving average in 2 dimensions

We define a self-affine surface $f(i,j)$ of size N,N, with $i,j \in [1,N]$. For each point $i,j \in [1,N-n+1]$ in this surface we define a subsurface of size $n \times n$, where each subsurface consists if the points

$$(k,l) \in \Big\{(i,j) + \big([1\ldots,n],[1,\ldots,n]\big)\Big\}$$

in the main surface. This means that the point (i,j) is located in the "lower left" corner of the subsurface, that the subsurfaces overlap, and that they together span the whole main surface. The limits $i,j \in [1,N-n+1]$ are set so that all subsurfaces are *inside* the main surface.

For each subsurface located at (i,j) we find a point (k_m,l_m) in the subsurface, which can be written as

$$(k_m,l_m) = (i,j) + (n-m,n-m), \tag{4.1}$$

where m is defined as

$$m = \lfloor n\theta \rfloor \qquad\qquad \text{for } \theta \in [0,1).$$

The parameter θ controls the position of this point inside the subsurface, and we have three extreme cases, listed below, and illustrated in fig. 4.2.

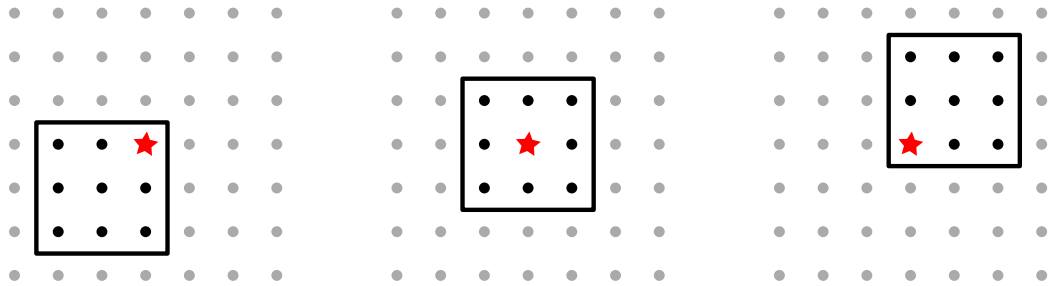
- $\boldsymbol{\theta = 0}$**:** the point (k_m,l_m) is in the upper right corner of the subsurface, at $(k_m,l_m) = (i+n,i+n)$
- $\boldsymbol{\theta = 1/2}$**:** the point (k_m,l_m) is in the center of the subsurface, at $(k_m,l_m) = (i+n/2,i+n/2)$
- $\boldsymbol{\theta \to 1 \to (n-1)/n}$**:** the point (k_m,l_m) is in the lower left corner of the subsurface, at $(k_m,l_m) = (i,j)$

We find the average $\bar{f}_n$ of each subsurface using

$$\bar{f}_n(i,j) = \frac{1}{n^2}\sum_{k=i}^{i+n}\sum_{l=j}^{j+n} f(k,l), \tag{4.2}$$

and we define the *generalized variance*, σ^2_{DMA}, as the sum of the squared differences between the value in the point $f(k_m,l_m)$ minus the average $\bar{f}_n(i,j)$, for each subsurface. This can be written as

$$\begin{aligned}\sigma^2_{\mathrm{DMA}} &= \frac{1}{(N-n)^2}\sum_{i=1}^{N-n+1}\sum_{j=1}^{N-n+1}\big(f(k_m,j_m) - \bar{f}_n(i,j)\big)^2 \\ &= \frac{1}{(N-n)^2}\sum_{i=1}^{N-n+1}\sum_{j=1}^{N-n+1}\big(f(i+n-m,j+n-m) - \bar{f}_n(i,j)\big)^2. \end{aligned} \tag{4.3}$$



(a) $\theta = 0$ **(b)** $\theta = 1/2$ **(c)** $\theta = (n-1)/n$

Figure 4.2: Illustration of three extreme cases for the parameter θ in DMA, on a surface. The dots are points where the main surface is defined, the red star is the point (k_m, j_m), and the black square marks the subsurface, and the points averaged over to calculate $\bar{f}_n(i,j)$ in eq. (4.2). The illustrations use $n = 3$.

It can be shown that this generalized variance has a power-law dependence on n [2, 6], which goes as

$$\sigma^2_{\mathrm{DMA}} \sim \left(2n^2\right)^H .$$

We can use this dependence to estimate the Hurst exponent H, by calculating σ^2_{DMA} for different sizes of the subsurfaces, n. We estimate H by a linear fit of $\log\left(\sigma_{\mathrm{DMA}}\right)^2$ against $\log\left(2n^2\right)$, where H is the slope of this fit.

In the paper by Anna Carbone that generalizes DMA to higher dimensions[6] they use different parameters for each spatial dimension d, $\boldsymbol{\theta} = \theta_1, \ldots, \theta_d$ and $\boldsymbol{n} = n_1, \ldots, n_d$, but for simplicity and to avoid spurious results, we use $\theta_1 = \theta_2 = \theta$ and $n_1 = n_2 = n$.

A modification of the method mentioned is replacing $\bar{f}_n(i,j)$ in eq. (4.3) with

$$\bar{f}_n{}^*(i,j) = (1-\alpha) f_n(i,j) + \alpha \bar{f}_n(i-1, j-1),$$

where

$$\alpha = n^2/(n+1)^2,$$

and $\tilde{f}_n(i,j)$ has the same form as before (see eq. (4.2)). To use this modification we also have to change the limits in eq. (4.3) from $i, j \in [1, N-n+1]$ to $i, j \in [2, N-n+1]$. This modification has been shown to give better results for small systems[6], and since we are usually generating surfaces with a resolution of 100-200 points in each direction, we use this modified method in our implementation of the method.

4.2.2 Validation

To verify that the method we used for estimating the Hurst exponent worked as intended, and gave good results, we ran a series of tests using synthetic 1-dimensional timeseries and 2-dimensional surfaces of fractional Brownian motion, with a known Hurst exponent. When doing these tests we soon realized that a big problem with synthesizing time series and surfaces with a given Hurst exponent is that it is both hard to accurately measure the exponent, and it is hard to synthesize data with a given exponent.

To test the DMA method we synthesized data with a given Hurst exponent using 4 different programs, and measured the exponent using the detrending moving average method for each of these methods.

For synthesizing 1D data we used the built-in Matlab-function `wfbm` which uses a wavelet-based synthesis method described by Abry and Sellan [1], and two methods from the Matlab-toolbox FracLab[12], `fbmwoodchan` which uses a method proposed by Wood and Chan in [49], and `fbmlevinson` which uses Cholesky/Levinson factorization from [22].

There are many methods and algorithms for generating surfaces data with a given Hurst exponent, but we had problems finding working implementations of any of them. We will later implement a midpoint displacement method for this (see chapter 5 and section 5.2.3), but having external reference is very useful, so we used a function from FracLab called `synth2`, which is not very well documented, but at least seems to generate accurate samples of fractional Brownian surfaces.

See fig. 4.3 for a plot of Hurst exponent measured using DMA as a function of the exponent used as input for the four different methods for generating synthetic data above, for three different values of the parameter θ. From the plots we conclude that $\theta = 0.0$ seems to give the best and most consistent results, as also noted by Gao-Feng Gu and Wei-Xing Zhou in [15], where they further develop the DMA method to analyse multifractals.

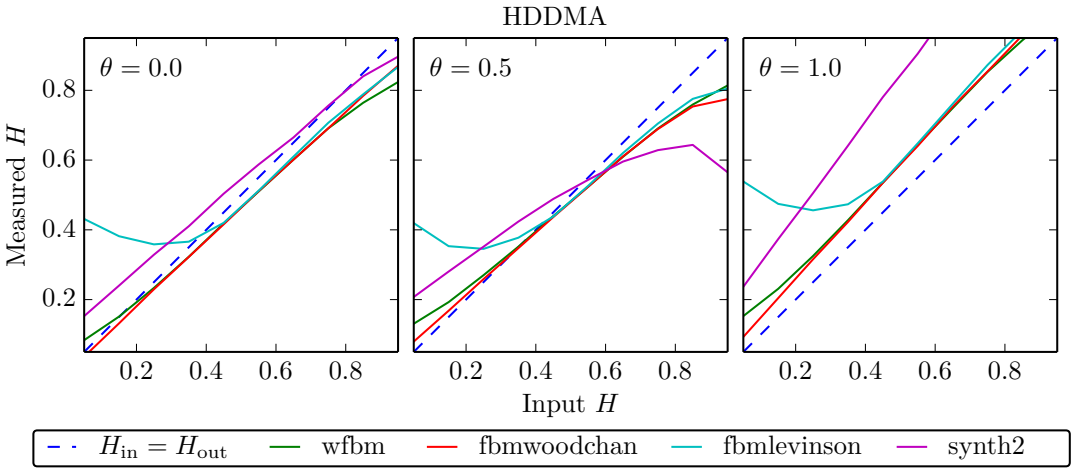


Figure 4.3: Plot of the Hurst exponent against the exponent used as input when generating the signals, as estimated by the detrending moving average method, used on data from four different synthetic signals, and for three different values of the parameter θ used in DMA. For the 1d methods we have averaged over 1000 samples for each point, and for `synth2` we have averaged over 100 samples, for input Hurst exponents between 0.05 and 0.95 in steps of 0.1. All methods except `synth2` generate 1-dimensional signals, while `synth2` generates a 2-dimensional signal.

Chapter 5

Generating surfaces and fractures

To generate random surfaces we use an iterative midpoint displacement method usually called successive random additions (SRA). The method is based on a method proposed by Fourner in 1982[11], but with some modifications suggested by Voss[46, 47]. The method has further been discussed by Saupe[36], amongst others. We choose this method mainly because it is possible to generate periodic surfaces with it, because it generates very good approximations to fBm surfaces[50], and because the Hurst exponent of the generated surfaces is easy to control. The method is also easy to understand, easy to implement, and generates surfaces with high resolution very fast. The method is widely used in scientific applications because of these properties, and is also used for generating surfaces in computer graphics, since the surfaces look very realistic.

5.1 Midpoint displacement methods

The method we use to generate random surfaces is very similar to the standard midpoint displacement method (MDM), so we start with showing that method. In 1 dimension this method goes as follows

1. Give the values at the endpoints of the interval, y_0 and y_n, random values from a Gaussian random variable with mean $\mu = 0$ and variance σ_0^2. This initial standard deviation σ_0 can be chosen freely.

2. Generate the value in the center of the interval, $y_{n/2}$, by averaging over the two endpoints and adding a Gaussian random number with mean $\mu = 0$

and a *reduced* variance

$$\sigma_1^2 = \big(1/2\big)^{2H} \sigma_0^2, \tag{5.1}$$

where H is the wanted Hurst exponent.

3. Continue generating the values in the center of each sub-interval until you reach the desired number of points, while reducing the variance of the random number by a factor $\frac{1}{2}$ each iteration. For iteration i we have

$$\sigma_i^2 = \big(1/2\big)^{i2H} \sigma_0^2. \tag{5.2}$$

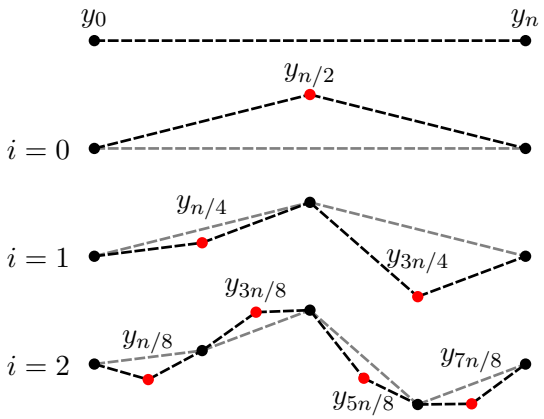


Figure 5.1: Illustration of the midpoint displacement method in 1 dimension. We increase the number of points from 2 to 9 using 3 iterations.

See fig. 5.1 for a visual illustration of the method.

This method generates a 1-dimensional line, with a Hurst exponent to the input H. But since we only add random numbers to the *new* values we generate each iteration, the result is non-stationary for $H \neq 0.5$[46], and it is neither truly self-similar or isotropic, as noted by Mandelbrot[26]. To mitigate this we implement the addition suggested by Voss[46], which consists of adding a random number to *all* points in each iteration, both the new and old. Voss called this modified method *successive random additions*.

5.2 Successive random additions

The method called *successive random additions* (SRA) is a modification of the regular midpoint displacement method first described by Richard F. Voss in [46], where we add random numbers to *all* points in each iteration, compared to just

adding random numbers to the *new* points in the regular midpoint displacement method. This modification means that we can replace the factor (1/2) in eqs. (5.1) and (5.2) with a general parameter r,and we get the following variance for iteration i

$$\sigma_i^2 = r^{2H}\sigma_{i-1}^2.$$

This new parameter r controls the lacunarity of the surface, without affecting the Hurst exponent.

5.2.1 Infinite grids

Voss has geneneralized the the method of successive random additions to higher dimensions[46], and this generalized form is the algorithm we use when generating fractures. We use the method to generate surfaces in the form of heightmaps, meaning a 2-dimensional grid of points (i, j), with a value for the height in each point, $z(i, j) = z_{i,j}$, which is generated by the algorithm.

The central part of the algorithm consists of two steps often called the *diamond-step* and the *square-step*. We start with a simple case of an infinite grid of evenly spaced points, all with known z-values. The two steps are as follows:

The *square-step*: The grid can be divided into small squares consisting of four points in each square, as in the leftmost square in fig. 5.2a. We generate the z-value in the center of each of these squares by averaging the z-values of the four corners of each square, as indicated by the red dots and arrows in fig. 5.2a. Then add a random Gaussian number with mean $\mu = 0$ and variance $\sigma_n^2 = \sigma_{n-1}^2 r^{2H}$ to all new and old points, where σ_{n-1}^2 is the variance used in the previous step of the algorithm.

The *diamond-step*: After the square-step the grid can be divided into smaller squares that are tilted by 45 degrees, as in the leftmost square in fig. 5.2b. We generate the z-values in the center of each square by averaging the z-values of the four corners of each square, as indicated by the red dots and arrows in fig. 5.2b. We then add a random Gaussian number with mean $\mu = 0$ and variance $\sigma_{n+1}^2 = \sigma_n^2 r^{2H}$, to all new and old points.

See fig. 5.3 for an illustration of the square-step and diamond-step applied once on a larger grid.

We see that by first applying the square-step and then applying the diamond-step, we add one point in between each point in each direction, almost doubling the resolution of the grid. In general we go from N to $N + (N - 1)$ points in each

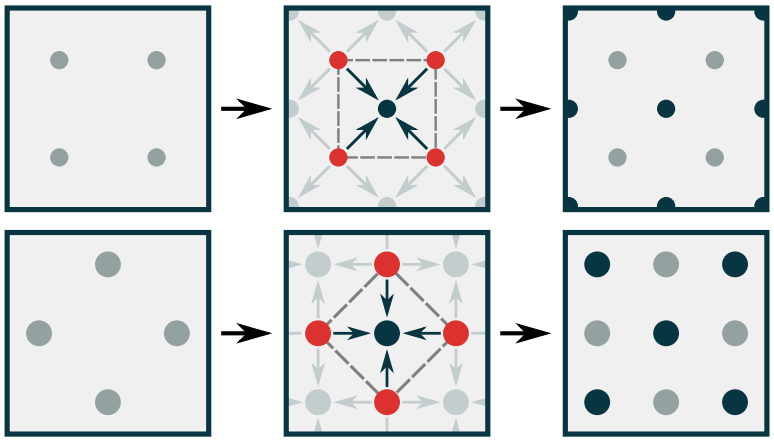


Figure 5.2: Illustration of the two steps used in the diamond square algorithm for generating random surfaces. The grey points are old points, the black points are new points, and the red points are the points used in the calcuation of the averages when generating the new points.

direction. By applying the algorithm several times we get

$$\begin{aligned} N_1 &= N_0 + (N_0 - 1) = 2N - 1 \\ N_2 &= 2N_1 - 1 = 4N_0 - 3 \\ N_3 &= 2N_2 - 1 = 9N_0 - 7 \\ &\vdots \\ N_n &= 2^n(N_0 - 1) + 1, \end{aligned} \tag{5.3}$$

where n is the number of times we have applied the algorithm, and N is the number of points in each direction. This means that using the diamond-square algorithm we can go from any resolution N_0 to all resolutions satisfying $N_n = 2^n(N_0-1)+1$, although if starting with points generated using a different method we do not have the same control over the Hurst exponent of the surface after generating new points.

5.2.2 Finite size effects

Since we are using computers to generate our surfaces, which have limited memory, we can not use infinite grids. This means we get some special cases that needs to be taken care of when generating points near the edges of the grid.

By applying the square step we generate one new point in the center of each square formed by the grid from the previous iteration, and in general we generate

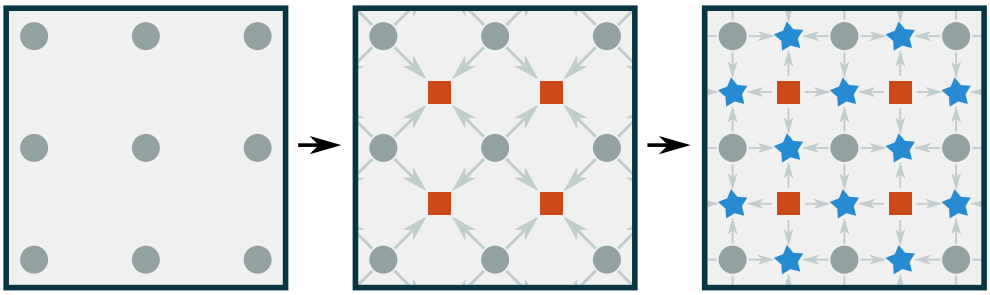


Figure 5.3: The diamond-square algorithm applied once on a grid of 3×3 points, increasing the number of points from 9 to 25. The orange square points are generated by the square-step (see fig. 5.2a), and the blue star-shaped points by the diamond-step (see fig. 5.2b).

the z-values in the points

$$z(i + 1/2, j + 1/2) \qquad i, j \in [0, N_{n-1}), \tag{5.4}$$

where (i, j) are the indices of the points in the grid after the previous iteration, and N_{n-1} is the number of points in each direction in this grid. In general the averages we calculate for the new points can be written as

$$z(i + 1/2, j + 1/2) = \frac{1}{4}\Big[z(i, j) + z(i + 1, j) + z(i, j + 1) + z(i + 1, j + 1)\Big], \tag{5.5}$$

using the limits in eq. (5.4). We see that the square-step only uses points inside the grid when calculating the averages, which means that we do not have to modify this step when going to a finite grid.

By applying the diamond-step we generate the values in the points

$$z(i + 1/2, j) \quad \text{for} \quad i \in [0, N_{n-1}) \quad \text{and} \quad j \in [0, N_{n-1}] \tag{5.6}$$

$$z(i, j + 1/2) \quad \text{for} \quad i \in [0, N_{n-1}] \quad \text{and} \quad j \in [0, N_{n-1}), \tag{5.7}$$

and in general the averages we calculate for the new points can be written as

$$z(i + 1/2, j) = \frac{1}{4}\Big[z(i, j) + z(i + 1, j) + z(i + 1/2, j - 1/2) + z(i + 1/2, j + 1/2)\Big] \tag{5.8}$$

$$z(i, j + 1/2) = \frac{1}{4}\Big[z(i, j) + z(i, j + 1) + z(i - 1/2, j + 1/2) + z(i + 1/2, j + 1/2)\Big], \tag{5.9}$$

using the limits in eqs. (5.6) and (5.7). We find that when generating points near the edges of the surface using the diamond-step, specifically when generating the points along the top and bottom edge

$$z(1/2, j) \qquad \text{and} \qquad z(n-1/2, j) \qquad \text{for} \qquad j \in [0, N_{n-1}],$$

and the points along the left and right edge

$$z(i, 1/2) \qquad \text{and} \qquad z(i, n-1/2) \qquad \text{for} \qquad i \in [0, N_{n-1}],$$

we need the values of points that lie outside the grid to calculate the averages. There are two possible solutions to this, that will generate different surfaces. If we want to generate a periodic surface, the solution is to wrap around the edges using periodic boundary conditions, and find the point we need on the opposite side of the grid. For example (using $i = j = 0$)

$$z(1/2, -1/2) \to z(1/2, n-1/2).$$
$$z(1/2, -1/2) \to z(1/2, n-1/2)$$

If generating a non-periodic surface we simply ignore the points that lie outside the grid when calculating the averages, and just calculate the average of the three other points.

5.2.3 Implementation

In our implementation we generate a surface on a finite grid of size $N \times N$, starting with only the z-values in the four corners defined, giving a resolution $N_0 = 2$. As shown in eq. (5.3) the algorithm can go from any resolution N_0 to any resolution $N_p = 2^p(n_0 - 1) + 1$ by applying the algorithm p times, which means that our implementation can generate surfaces with resolutions

$$N = 2^p(2-1) + 1 = 2^p + 1,$$

where p is any positive integer.

We implement generation of both periodic and non-periodic surfaces using using eqs. (5.4) to (5.9), while skipping points outside the grid for non-periodic surfaces, and wrapping around the edges using periodic boundary conditions when generating periodic surfaces.

To ensure that the periodic surfaces actually turn out periodic we start with all four corners having the same value. We also let the right and bottom edge be equivalent to the left and top edge, respectively, which effectively means that all four corners should always have the same z-value. To ensure that the opposite

edges stay equal to each other we never generate any points on the right and bottom edge, but just copy the z-values from the opposite edge after the diamond-step.

This leaves us with the following algorithm for generating a surface which approximates a 2-dimensional fractional Brownian motion with Hurst exponent H

1. Allocate a grid of size $N \times N$, where $N = 2^p + 1$, and p is any positive integer. This grid will store the z-values, or the height of the surface, in each grid point $z(x, y)$.
2. Initialize the z-values of the corners of the grid by drawing random numbers from a Gaussian distribution with mean $\mu = 0$ and variance σ_0. The initial variance can be chosen freely. The initial resolution is now 2×2.
   - If generating a periodic surface, give all four corners the same z-value.
3. Apply the square-step using eqs. (5.4) and (5.5). Add a random Gaussian number with mean $\mu = 0$ and variance $\sigma_n = \sigma_{n-1}^2 r^{2H}$ to all new and old points.
4. Apply the diamond-step using eqs. (5.8) to (5.9). Add a random Gaussian number with mean $\mu = 0$ and variance $\sigma_{n+1} = \sigma_n^2 r^{2H}$ to all new and old points.
   - If generating a periodic surface, skip generating z-values for points on the right and bottom edge using the diamond-step, and instead copy the values from the opposite edges after the diamond-step.
5. Repeat step 3 and 4 p times until you reach the desired resolution of $N \times N$, where $N = 2^p + 1$. For step n the variance of the random Gaussian numbers is

$$\sigma_n^2 = \sigma_0^2 (r^n)^{2H}.$$

We implement the method in `C++`, and make a `Matlab` interface to the `C++`-program, for fast visualization and testing. We implement both generation of periodic and non-periodic surfaces, and the midpoint displacement method, and successive random additions.

See fig. 5.4 for a surface with resolution 33×33 generated by the algorithm.

5.2.4 Validation

To test the method for generating random surfaces, and to check that the Hurst exponent of the surfaces correspond to the wanted exponent, we generate surfaces

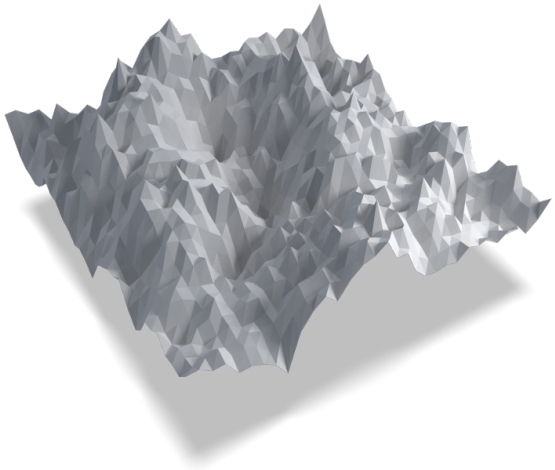


Figure 5.4: A surface with resolution 33×33 created using the midpoint displacement method called successive random additions.

and measure the Hurst exponent using the detrending moving average method from section 4.2. We have implemented both the midpoint displacement method and successive random additions for generating random surfaces, both for periodic and non-periodic surfaces. A plot of the measured Hurst exponent (using DMA) as function of the input Hurst exponent can be seen in fig. 5.5.

From the plot in fig. 5.5 we see that surfaces with periodic boundaries generally get a lower measured H than non-periodic surfaces, for surfaces with $H > 0.5$. We also see that the surfaces generated using the midpoint displacement method (MDM) have very similar Hurst exponents as the ones generated using successive random additions (SRA). We see that the Hurst exponent of all surfaces is generally lower than the input exponent for input $H > 0.5$, and lower than the input exponent for input $H < 0.5$. We should take note of this when generating surfaces for our experiments, and make sure to measure the actual exponent of the surfaces, since we see that the standard deviation is relatively high.

5.3 Generating fractures from surfaces

To generate a realistic fracture we use the method of successive random additions described in chapter 5 and section 5.2.3 to generate random surfaces with a known Hurst exponent. We then displace the surfaces in the z-direction, so one is above

[1] In reality we can not have Hurst exponents greater than 1, but as we see, the midpoint displacement methods generally creates surfaces with a measured exponent (H_{out}) lower than the input exponent (H_{in}) for $H_{\text{in}} > 0.5$, so to we use some samples of $H_{\text{in}} > 1$.

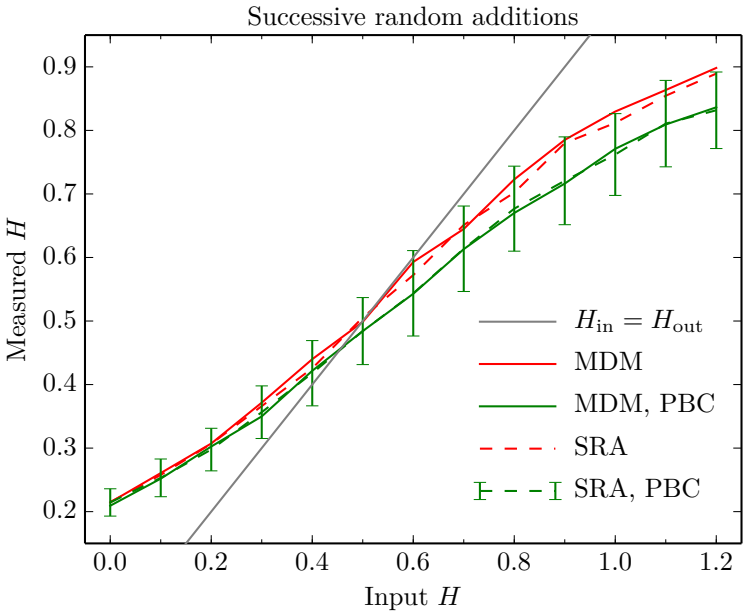


Figure 5.5: Plot of the Hurst exponent measured using detrending moving average (DMA), as function of the input Hurst exponent to the synthesizing method. The dashed grey line indicates a measured Hurst exponent of 0.75, the solid grey line a measured exponent exactly equal to the input exponent ($H_{\text{in}} = H_{\text{out}}$). The green lines are for surfaces created using periodic boundary conditions (PBC), the red lines using non-periodic boundaries, the dashed lines using successive random additions (SRA), and the solid lines using the regular midpoint displacement method (MDM). We used 100 samples for each point, and input Hurst exponents between 0 and 1.2[1] in steps of 0.1. We have plotted the standard deviation in each point for SRA with periodic boundary conditions, and the standard deviation is about the same for the other combinations of methods and boundary conditions.

the other, and let the space between the surfaces be the void[2].

In practice we make a fractured silica structure using the following procedure

- Prepare a slab of amorphous SiO_2.
- Generate two surfaces.
- Rescale the (x, y)-positions of surfaces so they span the molecular system.
- Rescale the z-values of the surfaces so all points are inside the system.

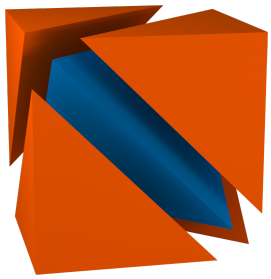
[2]Since we are using periodic systems, we could also have let the space outside the surfaces be the fracture and get the same result. But for easier visualization and understanding we use the volume between the surfaces.

- Remove all atoms between the upper and lower surface.

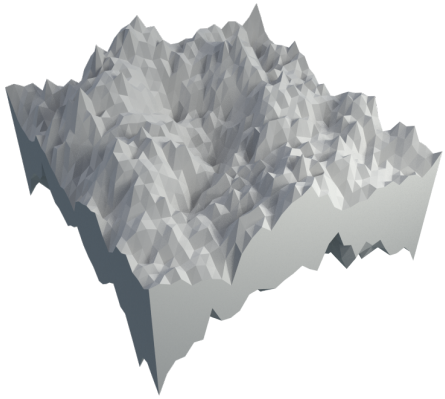
The biggest problem with this procedure is removing the atoms between two surfaces. But since all points in both surfaces lie on a regular grid in the x-y-plane, there is a simple way of dividing the volume between the surfaces into tetrahedra. And checking if a point is inside a tetrahedra is a geometrical exercize that can be solved programmatically. If the two surfaces are not intersecting, we can divide the volume between them into convex hexahedra spanned out by the points

$$z(i,j), \quad z(i+1,j), \qquad z(i,j+1), \quad \text{and} \quad z(i+1,j+1) \qquad \text{for} \quad i,j \in [0,N)$$

in the two surfaces (four points in each surface). We then divide each convex hexahedra into five tetraheda, as illustrated in fig. 5.6a, giving a total of $5(N-1)^2$ tetrahedra spanning the total volume between the two surfaces.



(a) Illustration of how to divide a convex hexahedron into five tetraheda.



(b) A random fracture made from two periodic surfaces.

5.3.1 Finding a point inside a tetrahedron

A tetrahedron consists of four points $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, and $\boldsymbol{d}$, and four faces spanned by the four possible combinations of the four points. For a face spanned by the points $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ we can find if a point $\boldsymbol{P}$ is on the same side of the face as the point $\boldsymbol{d}$ (the point not used to construct the face) by doing some geometry. We first find the normal vector to the surface $\boldsymbol{n}$ by the cross product

$$\boldsymbol{n} = (\boldsymbol{a} - \boldsymbol{c})(\boldsymbol{b} - \boldsymbol{c}).$$

We know that the sign of the dot product between this normal vector and another vector going from the plane to a point will give us information about which side of

the plane the point is. This means that if two points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are on the side of the plane, the dot product between the normal vector and the two vectors

$$(\boldsymbol{p}_i - \boldsymbol{k}),$$

where $\boldsymbol{k}$ is any point in the plane, should have the same sign. So we find the sign of dot products

$$\operatorname{sgn}\big(\boldsymbol{n} \cdot (\boldsymbol{P} - \boldsymbol{k})\big),$$
$$\operatorname{sgn}\big(\boldsymbol{n} \cdot (\boldsymbol{d} - \boldsymbol{k})\big),$$

and if the sign of these dot products is the same, we know that the point $\boldsymbol{P}$ is on the same side of the face as the point $\boldsymbol{d}$. We now see that if we do this for all four faces of the tetrahedra, we know that the point $\boldsymbol{P}$ is inside the tetrahedra if the signs of *all* pairs of dot products are equal.

To implement this for checking which atoms are between two surfaces (with the volume between the surfaces divided into tetrahedra), we express it as a matrix equation. This reduces the calculation of wether a point is inside a tetrahedron to comparing the signs of five matrix determinants.

Part III

Simulations

Introduction

In this chapter we will present the procedure we have used when doing simulations, the systems we have studied, and the the results we have found.

We start with a description of what steps we have used to generate a realistic nanoporous silica, with water in the pores. We describe how we create a random fracture in a slab of silica, and the method we have developed for passivating the dangling ends after we cut out the fracture. We also describe a method for injecting water into the fracture.

After this we present the different measurements we have done during the simulations, how the measurements are done, and why we do them. We then describe the different systems we have generated, the characteristics of these systems, both in form of tables, and using renderings of visualizations of the systems.

Finally the results from all our simulations and measurements are presented and discussed.

Chapter 6

Simulation procedure

When doing simulations using molecular dynamics we use a procedure akin that used by actual experiments. Since the duration of the simulations we are realistically able to simulate on are of the order of tens of nanoseconds, we have to be smart when initializing the system, to avoid having to simulate for a long time to get to the state we want to study. This means that we should start out with the system in a state as close to the one we want to study as possible. The problem with this when simulating silica is that the silica structure formed when rapidly cooling molten silica does not have any long-range ordering. Silica in the glass form has an amorph structure, which does not have any long-range ordering, but has short-range ordering well beyond the Si-O bond length. This structure is hard to set up with an algorithm.

6.1 Initialization

To generate silica in the glass form we first create a perfect silica crystal in the crystalline form β-cristobalite, the unit cell of which can be seen in figure fig. 6.1, and a larger crystal in fig. 6.3a. This crystalline form consists of corner-bonded SiO_4-tetrahedra, and in the perfect crystallic form all silicon atoms are bound to four oxygen atoms, and all oxygen atoms to two silicon atoms.

We give the atoms a random uniformly distributed velocities with mean $\mu = 0$ and standard deviation $\sigma \propto \sqrt{T}$, where T is the wanted temperature.

We then heat the system to 4500 K in steps of 700 K to melt the silica crystal. Since we are mainly interested in controlling the temperature at this stage, the Berendsen thermostat is used for these temperature changes. We alternate between using a thermostat to adjust the temperature, and simulating with the

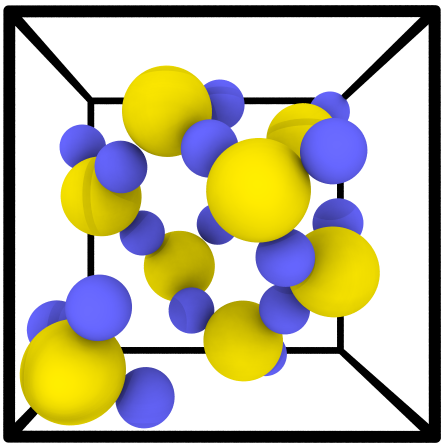


Figure 6.1: β-cristobalite unit cell, with 8 silicon atoms and 16 oxygen atoms.

thermostat off, to let the system thermalize and react to the temperature change after applying the thermostat. The number of timesteps we used for the thermostat period is around 2 500, and for the thermalization period around 10 000. We then cool the system by doing the previous procedure in reverse. See fig. 6.2 for a plot of the temperature as function of time while we melt and cool down the system, and fig. 6.3a for a visualization of a perfect crystal of β-cristobalite, and fig. 6.3c for amorphous, solid silica.

The initialization procedure is visualized in fig. 6.3, and can be summed up as follows

- Generate a perfect crystal of β-cristobalite of the wanted size (fig. 6.3a).
- Heat the system to well above the melting point of silica (we usually use 4 500 Kelvin, using a thermostat (Berendsen) (fig. 6.3b).
- Cool down the system to well below the glass-transition temperature (we use 300 Kelvin), using a thermostat (Berendsen) (fig. 6.3c).
- Cut out the fracture (fig. 6.3d).
- Passivate the dangling ends and apply steepest descent to let the passivation atoms find their optimal positions (fig. 6.3e).
- Fill the pore with water, and thermalize the system at 300 K (fig. 6.3f).

We now have a thermalized and (hopefully) realistic silica crystal at near room temperature. From this crystal we cut out the fracture, passivate using one of the passivation methods, fill the fracture with water molecules, and apply a simple steepest descent procedure to let the inserted atoms find their equilibrium positions. After filling the fracture with water we need to thermalize the system again, since the energy (and thereby the temperature) changes when we remove

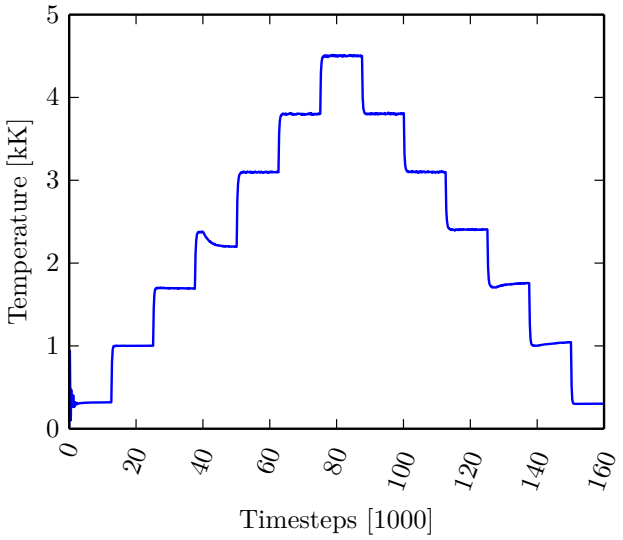


Figure 6.2: Plot of the temperature (in kilo-Kelvin) as function of timesteps when melting and cooling down a silica system, using the Berendsen thermostat. We use timesteps of 0.050 picoseconds, and use 2 500 timesteps with the thermostat turned on, and then 10 000 timesteps to let the system thermalize (with the thermostat off), for each step in temperature.

and insert atoms.

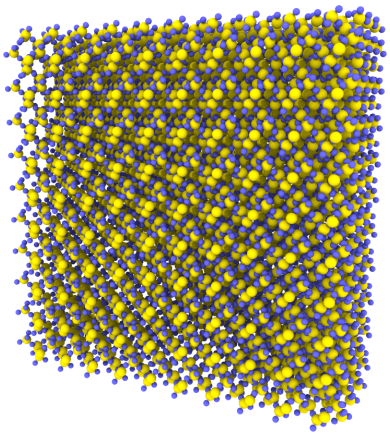
6.2 Passivation

In most silicates the silicon atoms have tetrahedral coordination, with four oxygen atoms surronding each silicon atom. When we remove silica- and oxygen-atoms to create a fracture, we do not take this into consideration. This means that we get dangling unsaturated bonds in the system, located near the surface of the pore. To rectify this we use a method called passivation, where we saturate and passivate the dangling bonds by inserting new atoms.

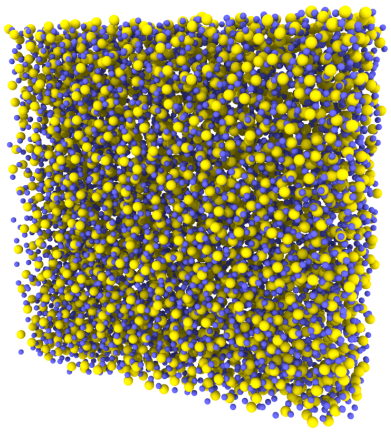
6.2.1 Water chemistry

Since we are going to inject water into the pore later on, we want to use the constituents of water to passivate the system. We know that water autodissociates into H^+ and OH^- via the following reaction

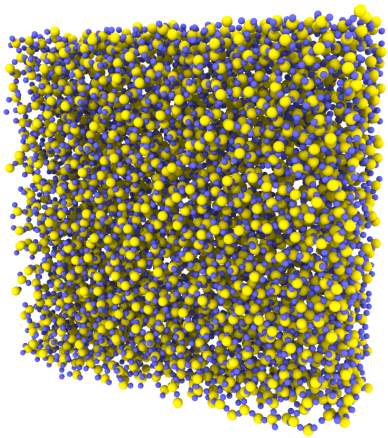
$$H_2O \rightleftharpoons H^+ + OH^-,$$



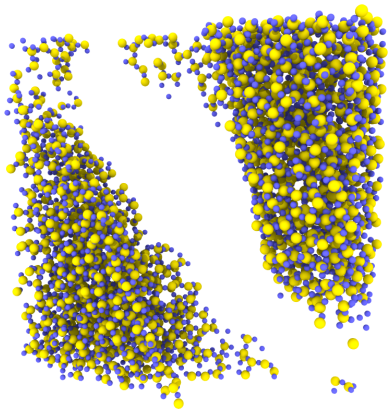
(a) Perfect β-cristobalite crystal.



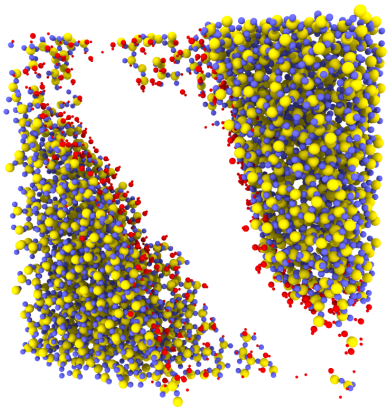
(b) Silica heated to 4500 K.



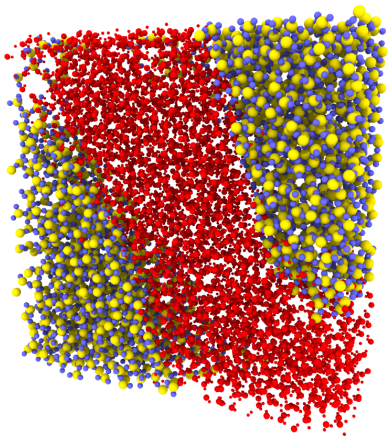
(c) Silica cooled to 300 K.



(d) Fracture cut out.



(e) Dangling ends passivated.



(f) Pore filled with water.

Figure 6.3: Visualizations of the different stages of initialization of a fracture in silica filled with water. We show a $75 \times 75 \times 25$ Å slice of a much larger system $(172 \text{ Å})^3$. The silicon atoms are yellow, the silica-oxygen blue, and hydrogen and water-oxygen red.

meaning that hydrogen (H) and hydroxide (OH) will be freely available in the system after filling the pore with water. On this background we choose to passivate the system using hydrogen and hydroxide. To avoid getting an acidic or alkaline system after the passivation procedure we should make sure to use equal parts hydrogen and hydroxide when passivating.

6.2.2 Passivating using hydrogen and hydroxide

After thermalizing our silica system we end up with a system consisting almost exclusively of SiO_2 tetrahedra. These tetrahedra are each formed by four oxygen atoms, one in each corner, and a silicon atom in the center. Each of these tetrahedra are then bonded to four other tetrahedra, by sharing the oxygen atoms in the corners. This way each oxygen atom is bonded to two silicon atoms, and each silicon atom to four oxygen atoms, giving an average chemical formula of SiO_2.

Since we do not take chemical bonds into consideration when removing atoms to create a fracture, we end up with some incomplete tetrahedra, with some silicon atoms bonded to less than four oxygen atoms, and some oxygen atoms bonded to less than two silicon atoms. See fig. 6.4 for an illustration of three different incomplete tetrahedra. This creates what we call dangling ends or unsaturated bonds, which we want to passivate.

To passivate the silicon atoms that are bonded to less than four oxygen atoms, we see that we need to complete the incomplete SiO_4 tetrahedra that have been created in the system. But if we only insert oxygen atoms in the positions of the missing oxygen atoms, we end up with new dangling ends, since the inserted oxygen atoms will only be bonded to one silicon atom. But, as we just saw, we will have hydroxide (OH) groups available in the system after filling the fracture with water. So instead of inserting oxygen atoms and creating new unsaturated bonds, we insert hydroxide groups and create saturated Si-O-H bonds. We put the hydrogen atom so that the Si-O-H angle is close to the angle in water molecules, 107.5 degrees.

To passivate the oxygen atoms that are bonded to only one silicon atom, we can use the hydrogen atoms that are avilable after filling the fracture with water, turning unsaturated SiO-groups into the same saturated Si-O-H-groups as before. We here too insert the hydrogen atoms with the Si-O-H angle close to 107.5 degrees.

In total we use the following procedure to passivate a system after creating a fracture:

- Remove all silicon and oxygen atoms that are not bonded to any atoms, since they are essentially not part of the silica matrix.
- Add one hydrogen atom to all oxygen atoms bonded to only one silicon atom. The hydrogen atoms are inserted approximately 0.95 Å from the oxygen atoms, with the hydrogen atom pointing away from the silicon atom, and with the Si-O-H angle close to 107.5 degrees.
- Add $(4 - n)$ hydroxide groups to silicon atoms bonded to $(1 \leq n < 4)$ oxygen atoms. We assume that the most stable position for the oxygen in the hydroxide groups are close to the tetrahedral positions of the missing oxygen atoms, and insert the hydroxide groups in these positions. The hydroxide groups are inserted approximately 1.65 Å from the silicon atoms, measured from the position of the silicon atom to the oxygen atom in the hydroxide groups, with the hydrogen atom pointing away from the silicon atom, and with the Si-O-H angle close to 107.5 degrees.

The lengths used are approximate experimental lengths found in naturally occuring silanols and water (see [23] for the Si-O length in silanol, and [7] for the O-H length in water). This procedure turns all dangling ends into stable, passive silanol groups.

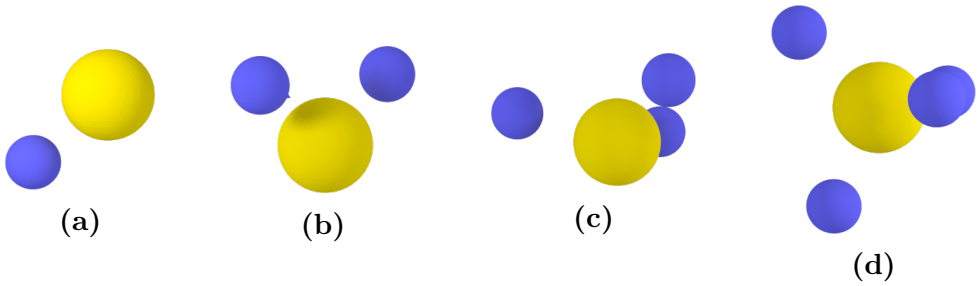


Figure 6.4: Illustration of four different incomplete silica tetrahedra, with respectively one, two, three and no missing oxygen atoms (**(d)** is a complete silica tetrahedra).

6.2.3 Counting number of bonds

Since we do not have actual bonds in molecular dynamics simulations, we do not know which atoms are bonded to which. So to find the number of bonded atoms for each silicon and oxygen atom, we create what we call *neigbor lists*. These neighbor lists are a list of atoms within a chosen radius for each atom. To create these lists we use the procedure detailed in section 7.2.1. Since we only have silicon and oxygen atoms in our system, we only need to specify a maximum the Si-O-distance to find which atoms are bonded. If we choose this distance

properly, we should be able find a good approximation to how many atoms each atom is bonded to.

6.2.4 Only passivating surface atoms

When implementing the passivation method detailed above, we soon ran into problems with silica and oxygen atoms that were bonded to too few atoms according to our rules above, while counting the number of bonds using a fixed radius. Some improvements were made by fine-tuning the radius used for each atom type, but we still often ended up passivating atoms that were inside the silica matrix, where we should not have any dangling bonds. To avoid this we came up with a method to only passivate the atoms at or near the surface of the fracture.

To do this we yet again use the voxelation method from 7.2, but this time we use a voxel size of around 6 Å. We then mark all voxels with atoms in them as occupied. We now see that if we find all *occupied* voxels with at least one *unoccupied* neighbor voxel (using 26-neighbor connectivity), we should have a list of the voxels that make up the surface of the fracture, and these voxels then contain all atoms at or near the surface of the fracture. We then use this list of atoms as input to the passivation program, and only passivate atoms in that list. See fig. 6.5 for an illustration of the method that finds the voxels and atoms at the surface of the fracture.

6.2.5 Passivation examples

An example of a system after passivation can be seen in fig. 6.6, where we have colored the passivating oxygen and hydrogen atoms red.

A good measure of the performance of the passivation method is the surface density of silanol after passivation. This number is often called the *silanol number*, and us considered to be a physico-chemical constant, with a numerical value $\alpha_{\mathrm{OH}} = 4.6$ (least-squares method) and α_{OH} nm^{-2} (arithmical mean) [51], and is known in literature as the Kiselev-Zhuravlev constant. As we will see, measuring the surface area of porous system is not trivial, so estimating this density is not trivial. But by creating a completely flat pore and passivating it, we found that we got a silanol surface density between 4 and 7 nm^{-2}, depending on how we measure the surface area of the pore, and how we cound the number of silanol groups.

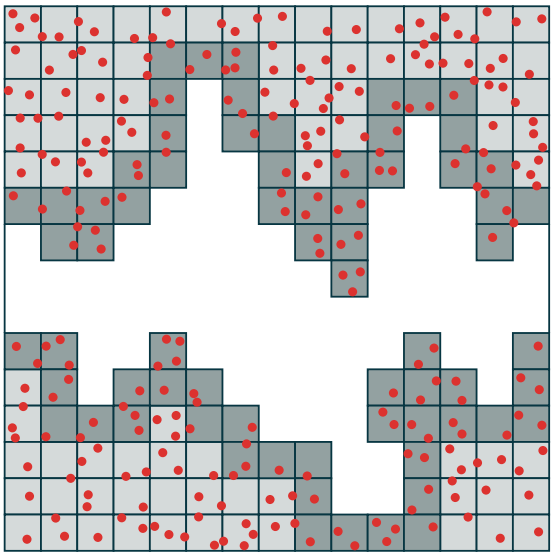


Figure 6.5: Illustration of a method for finding atoms and voxels at the surface of a fracture. All gray voxels are occupied voxels (with at least one atom in them), and the dark gray voxels are the voxels with at least one unoccupied neighbor voxel.

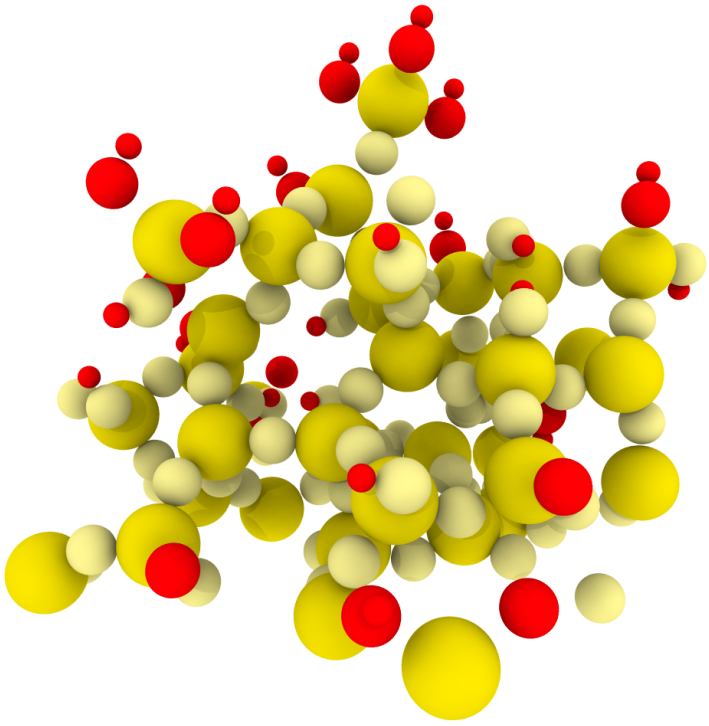


Figure 6.6: Example of the result of the passivation procedure. Here the oxygen and hydrogen molecules are red, silicon atoms are yellow, and silicon-oxygen atoms are light yellow.

6.3 Injecting water

After removing atoms to create a fracture, and passivating the system, we are now ready to inject water into the fracture. To do this we use the technique of *voxelation* (see section 7.2). We first divide the system into voxels, find the voxels that make up the void, and then put water molecules in these voxels. The water density can then be controlled by the size of the voxels we use, and how many of the voxels we fill.

6.3.1 Finding correct voxel size

If we want to inject water with density ρ, we can find the voxel size we need from the molar mass of water, $M_{\mathrm{H_2O}} = M = 0.0180158$ kg/mol. We use the molar mass and wanted density to find the "volume" each water atom should occupy, as follows

$$V = \frac{M\ [\mathrm{kg/mol}] \times \dfrac{1}{N_A\ [\mathrm{mol}^{-1}]}}{\rho\ [\mathrm{kg/m^3}]} = \frac{M}{\rho N_A}[\mathrm{m^3}]$$

where N_A is the Avogadro constant. From here we find the size of the voxels by taking the cube root

$$l = \left(\frac{M}{\rho N_A}\right)^{1/3}\ \mathrm{m}. \tag{6.1}$$

To get a water density approximately equal to ρ we can then use a voxel size l, and put one water atom in each voxel. If we for example want to insert water with $\rho = 1000$ kg/m^3, approximately the density of water in room temperature, we get a voxel size of

$$l = \left(\frac{0.0180158\ \mathrm{kg/mol}}{1000\ \mathrm{kg/m^3} \times 6.0221 \times 10^{23}\ \mathrm{mol}^{-1}}\right)^{1/3} = 3.1\ \text{Å}.$$

Voxel size in finite systems

Since we have a finite system we usually can not use the exact voxel size we want, but we have to divide the system into an integer number of voxels. This means that we will not get the exact density we want if we fill all voxels. To remedy this we only fill the fraction of voxels to get the wanted density.

In practice we use the following procedure

- Find the number of voxels to divide the system into (in each direction) via

$$n_i = \left\lceil \frac{L_i}{l_i} \right\rceil,$$

where L_i is the system size in dimension i and l_i is the voxel size calculated using eq. (6.1). We use the ceiling-function to ensure that the actual voxel size we use is smaller than (or equal to) the voxel size we calculated. We find the actual voxel size via

$$\tilde{l}_i = \frac{L_i}{n_i}.$$

- Divide the system into voxels of size $(\tilde{l}_x, \tilde{l}_y, \tilde{l}_z)$, and find the voxels that make up the void (see section 6.3.2 for more on this).
- To find the ratio of voxels to put water molecules in we first calculate the density we would get if we filled all empty voxels using

$$\tilde{\rho} = \frac{M}{\tilde{l}_x \tilde{l}_y \tilde{l}_z N_A},$$

and then find the number of voxels to fill as

$$\tilde{N} = N\frac{\tilde{\rho}}{\rho} = N\frac{\tilde{l}_x \tilde{l}_y \tilde{l}_z}{l_x l_y l_z},$$

where N is the total number of empty voxels.

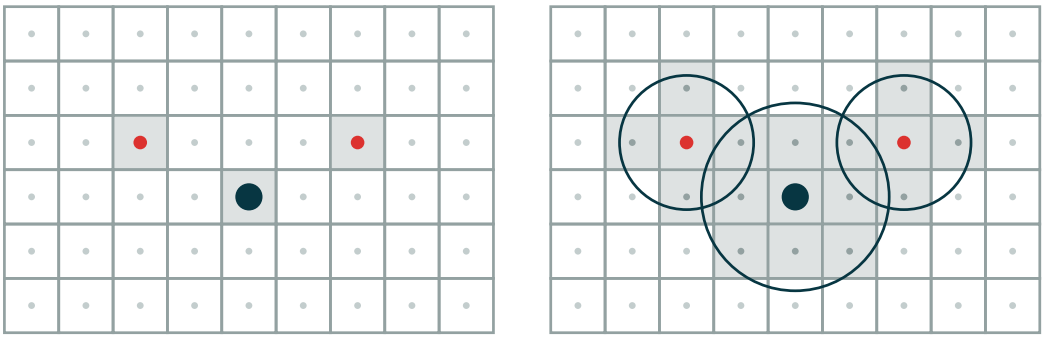
This can for example be done by looping over all voxels, drawing a random uniform number between 0 and 1 for each voxel, and putting a molecule in the voxel if the random number is smaller than $\tilde{N}/N$.

The water molecules are inserted with the oxygen atom in the center of the voxel, and the two hydrogen atoms pointing in a random direction, but with an angle of $\sim 104.45°$.

6.3.2 Identifying the voxels that make up the void

The naive way of finding the voxels to fill with water is to just find which voxel each silicon and oxygen atom is in, and mark those as occupied. Using this method we found that we often got some empty voxels inside the silica matrix, which meant we got single water atoms trapped inside what was supposed to be the silica matrix. This is most likely cause by the amorphous structure of silica in the glass state, in which there are small pores spread throughout the structure.

To solve this we assign a radius to each atom type, and mark all voxels with the center of the voxel within this radius from an atom as occupied. The rest of the voxels should now be a good approximation to the void. See fig. 6.7 for an illustration of this procedure.



(a) Marking only one voxel per atom as occupied.

(b) Marking all voxels within radius from atom as occupied.

Figure 6.7: To find voxels that make up the void/pore in we can either **a)** mark the voxel each existing atom belongs in as occupied, or **b)** mark all voxels within a radius from each atom as occupied. We can assign a different radius to each atom. We have illustrated using part of a silica tetrahedra, with one silicon atom (●) and two oxygen atoms (•). The center of each voxel is marked by a dot.

A different solution to the problem of tiny pores inside the silica matrix is to remove all small clusters of voxels (where a "small cluster of voxels" would need to be defined), or perhaps to use different voxel sizes for finding the void and filling the void with water.

Chapter 7

Measurements

We have now developed a molecular dynamics program that can do realistic simulations of water, silica, and model the interactions between water and silica. We have developed methods for initializing nanoporous silica systems, with randomly generated rough fractures in them, and methods for injecting water into these pores.

We know that the hydrophilic nature of silica creates some interactions between water and silica, and it is the effect of these interactions we want to study. We expect these interactions to have short-range effects on the water, so calculating averages for all water atoms seems like a bad approach, since the fine details of what happens with water near the silica surface will be lost. To be able to study the water-silica interface we thus need to find ways to find and do measurements on the water molecules near the silica surface.

7.1 Distance to silica matrix

To study the water-silica interface we want to do measures as function of the distance to the surface of the pore, in our case meaning the distance from a water molecule to the interface between water and silica. The first problem with this is to find out how to measure the distance from a point, for example a water molecule, to the surface. Most of our measures are done on water molecules in fractures and pores, so we first define the position of the water molecule as equal to the position of the oxygen atom in the water molecule. We then use the distance from these water-oxygen atoms to the nearest silica atom to define a distance from the water molecule to the surface of the silica matrix. Finding the nearest silica atom is not trivial though, so a separate procedure for doing this is shown in section 7.2.2.

When measuring quantities that only depend on data from one timestep we do not have to worry about that the atoms move, so we just sort the atoms by distance to the surface using the procedure in section 7.2.2 and do our measurements, individually on each timestep. But if we want to study for example diffusion, or the tetrahedral order parameter, which depend on data from several timesteps, we have to find a good way to define which atoms are in a certain range of the surface. We tried different methods, but ended up using the average distance to the surface for this.

7.1.1 Note on this method

When we define the distance to the silica matrix as the distance to the nearest silicon atom, we get some effects we should take note of for atoms very close to the matrix. With our definition of the distance to the silica matrix we are effectively making our bins out of spherical shells centered on each silicon atom. Compared to for example using the z-distance to the surface in a completely flat fracture, where we know the z-height of the surface, we see that this can give different results. The problem is of course that in a random fracture in a silica system we can not easily define a normal vector to the surface of the fracture, so finding an equivalent to the z-distance in such a fracture is hard.

When we do our measurements as a function of distance to the silica matrix we usually sort the atoms into bins according to their (average) distance to the nearest silicon atom. At distances much larger than the average distance between the silicon atoms at the surface of the fracture this is not a problem, since the curature of the spherical shells is low, and the volume enclosed by the shells is pretty close to the one we would have gotten if we had created bins used z-positions in flat fracture. But at distances close to the average distances between the silicon atoms we begin to see that the volumes our bins consist of start curving around the silicon atoms, instead of staying flat as they would have if we were using the z-distance. See fig. 7.2 for an illustration of this. In this illustration we have illustrated bins created using the same bin width and distance from the matrix, using the two different methods. The dark gray areas are the ones that are included using both methods ($A_{\mathrm{Si}} \cap A_z$), the light grey areas (A_{Si}) are the ones unique to the spherical shell bins, and the yellow areas are the ones unique to the the z-distance bins (A_z).

This difference in binning is something we should be wary about when comparing our results to measurements done using the z-distance as the distance to the matrix. One result of this can be seen in fig. 7.1, where we have plotted the number of atoms in each bin of width 0.25 Å against the distance from the silica matrix (meaning the radius of the spherical shells). We see that we get no atoms

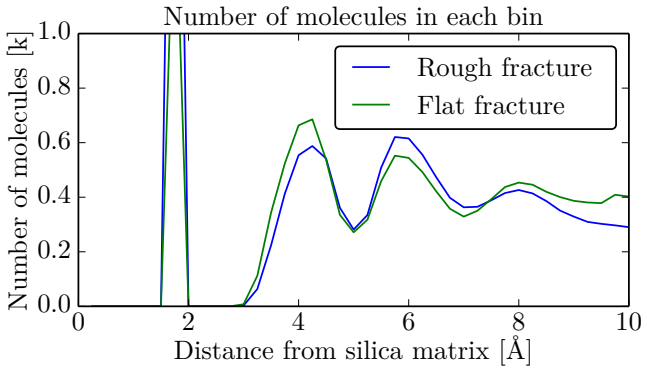


Figure 7.1: Plot of the number of atoms in each bin, when using the distance to the nearest atom for binning.

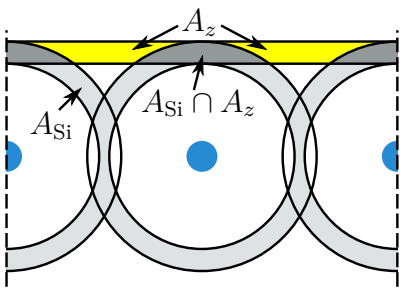


Figure 7.2: Illustration of binning when using distance to nearest silicon atom as definition of distance to silica matrix (r_{Si}), vs. z-distance (r_z).

in the bins from 2 to 3 Ångström, but a big spike just below 2 Å. This spike is most likely caused by water-oxygen atoms that are bound to silicon atoms, which we placed there when passivating the system (silicon-water atoms from the initial silica crystal are not included in the counting, but we use water-oxygen when passivating). If we had used the z-distance in a flat fracture in a system like this, we would most likely have gotten a very different distribution below 4 Å, with a more flat distribution instead of it going to zero, since the angular distribution of the oxygen atoms around the silica atoms makes the z-distance vary.

7.2 Voxelation

The method of voxelation is a method where we divide our simulation system into adjacent boxes or *voxels* (3-dimensional pixels). The system is divided into $n_x \times n_y \times n_z$ voxels of size $l_x \times l_y \times l_z$. Depending the what we want to calculate we can calculate the voxel size from the number of voxels, or vice versa, using the following relation

$$n_i = \frac{L_i}{l_i},$$

where L_i is the system size, and we assume that the voxel size l_i is set so that L_i is evenly divisible by l_i (meaning that the remainder of L_i/l_i is zero). If we have a maximum or minimum voxel size $l_i^{\max}$ or $l_i^{\min}$ we can use the following relations to calculate the number of voxels

$$n_i = \left\lfloor \frac{L_i}{l_i^{\max}} \right\rfloor \qquad \text{or} \qquad n_i = \left\lceil \frac{L_i}{l_i^{\min}} \right\rceil,$$

where $\lfloor x \rfloor$ is the `floor`-function and $\lceil x \rceil$ is the `ceil`-function.

The voxels are indexed (i, j, k) where $i, j, k \in [0, n_i - 1]$, and a voxel is defined as the points (x, y, z) where

$$\left\lfloor \frac{x}{l_x} \right\rfloor = i, \tag{7.1}$$

and similarly for the other dimensions.

7.2.1 Neighbor lists

When doing calculations and measurements on a molecular system, we often need information about the neighboring atoms of each atom, and we want to make a so-called *neighbor list*, which are lists of which atoms are within a distance dr of each atom. Finding out which atoms are within a certain distance of each atom can take a long time; the trivial way of checking each atom against all other atoms scales as $\mathcal{O}(N^2)$, N being the number of atoms.

Since we want to find all neighbors within a distance dr of a point, for all or most of the atoms, we can use the voxelation method to do it efficiently. To do this we first voxelate the system using a minimum voxel size equal to r. We then find which voxel each atom belongs to, and store this. We can then find the atoms within a distance dr from a point (x, y, z) by first finding the voxel this point lies in using (from eq. (7.1))

$$i = \left\lfloor \frac{x}{l_x} \right\rfloor, \qquad j = \left\lfloor \frac{y}{l_y} \right\rfloor, \qquad j = \left\lfloor \frac{z}{l_z} \right\rfloor,$$

where $l_x \times l_y \times l_z$ is the actual voxel size (we need an even number of voxels, so the actual voxel size is governed by the system size). We then check the distance between the point and the atoms in the voxel the point belongs in, and the atoms in the 26 neighboring voxels of this voxel.

Checking the 26 neighboring voxels ensure that we included all atoms within the distance dr. We can see this by looking at the worst case example, where we have a point right at the edge of the voxel it belongs to, at $(i + (1 - \epsilon_0))$, and an atom in voxel $(i + 2)$ being as close to the point as possible, at $(i + 2 + \epsilon)$. The distance between those two points would then be

$$\begin{aligned}((i+2)l + \epsilon_1) - (il + (l - \epsilon_0)) &= ((i+2) - (i+1) - \epsilon_0)l + \epsilon_1 + \epsilon_0 \\ &= l + \epsilon + \epsilon_0,\end{aligned}$$

which is larger than l, since ϵ_0, ϵ_1 have to be larger than 0.

When voxelating the system using the distance r we should take care not to use a too small distance, i.e. make the voxels too small and create a lot of voxels. Since the total number of voxels goes as n^3 the memory needed to store the matrix increases rapidly with decreasing voxel size. To avoid this we usually implement a hard limit to the number of voxels, and found that a limit of $n < 256$ or even $n < 128$ seemed to work good in most cases. On the other hand, if we make the voxels too large we soon find that the program is not especially efficient. This is because most voxels will have a lot of atoms in then, and we have to look through a lot of atoms when checking the 26+1 voxels for each atom.

An implementation of the voxelation method for creating neighbor lists can be seen in listing 7.1. Note that when calculating distances between points we usually calculate and compare squared distances like $r^2 = (x_1 - x_2)(x_1 - x_2) + \ldots$, since calculating roots are a time-consuming operation on a computer (at least compared to multiplication and addition).

7.2.2 Finding distance to surface

When doing measurements on water molecules we often want to know the distance from the water molecule to the surface of the pore the water molecule is in. To find this we first define the position of the water molecule as the position of the oxygen atom in the molecule. We then use the distance between this oxygen atom to the nearest silicon atom as the distance to the surface.

To use the voxelation method we need to have a maximum distance to look for silicon atoms in. This atom should be set as small as possible, to efficiently use the voxelation method[1]. We divide the system into voxels using the technique from section 7.2, and sort all silicon atoms into the voxels. For each water-oxygen atom we then find the distance to the nearest silicon atom by calculating the distance between the oxygen atom and the silicon atoms in the voxel the oxygen atom belongs in, and the silicon atoms in all 26 neighbor voxels. See section 7.2.1 for more details.

[1] We usually implement a hard upper bound on the number of voxels, or a lower bound on the voxel size, to keep the memory consumption of our program in check.

```
int nVoxels = floor(systemSize/radius);
double voxelSize = systemSize*nVoxels;

sortAtomsIntoVoxels(atoms, voxelSize, voxels);

vector<vector<Atom*> > neighborAtoms(atoms.size());

// Loop over all atoms
for (Atom *atom : atoms) {
    // Index of the voxel this atom belongs to
    ivec3 index = floor(atom.position() / voxelSize)

    // Loop over all 27 neighbor voxels (including self)
    for (int di = -1; di <= 1; di++)
    for (int dj = -1; dj <= 1; dj++)
    for (int dk = -1; dk <= 1; dk++)
    {{{
        // Index of neighbor voxel using periodic boundary conditions
        // nx, ny, nz is the number of voxels in each direction
        int i = (index[0] + di + nx) % nx;
        int j = (index[1] + dj + ny) % ny;
        int k = (index[2] + dk + nz) % nz;

        neighborAtoms[atom.index()].push_back(
            findAtomsWithinRadius(atom, voxels[i][j][k], radiusSquared)
        );
    }}}
}
```

Listing 7.1: An example of how to find the neighbor atoms within a given distance (`radius`) of all atoms. This example assumes a cubic system of size `systemSize`. See listings 7.2 and 7.3 for example implentations of `sortAtomsIntoVoxels` and `findAtomsWithinRadius`.

```
void sortAtomsIntoVoxels(
    const vector<Atom*> &atoms,
    double voxelSize,
    vector<vector<vector<Atom*> > > &voxels) {

    for (Atom *atom : atoms) {
        // Index of the voxel this atom belongs to
        int i = floor(atom.position().x() / voxelSize);
        int j = floor(atom.position().y() / voxelSize);
        int k = floor(atom.position().z() / voxelSize);
        voxels[i][j][k].push_back(atom);
    }
}
```

Listing 7.2: Example of implementation of `sortAtomsIntoVoxels` from listing 7.1, for sorting atoms into voxels with size `voxelSize`. We use the `floor` function to get the index of the voxel each atom belongs in, using zero-based numbering.

```
vector<Atom*> findAtomsWithinRadius(
    Atom *atom1, const vector<Atom*> &voxel, double radiusSquared) {

    vector<Atom*> neighborAtoms;

    // Loop over atoms in neighbor voxel
    for (Atom *atom2 : voxel) {
        if (atom2 != atom1) {
            double drSquared =
                calculateDistanceSquaredBetweenAtoms(atom1, atom2);
            if (drSquared < radiusSquared) {
                neighborAtoms.push_back(atom2);
            }
        }
    }
    return neighborAtoms;
}
```

Listing 7.3: Example implementation of `findAtomsWithinRadius` from listing 7.1. See listing 7.4 for an example implementation of `calculateDistanceSquaredBetweenAtoms`.

```
double calculateDistanceSquaredBetweenAtoms(Atom *atom1, Atom *atom2) {
    vec3 dr = atom2->position() - atom1->position();

    // Minimum image convention
    for (int dim = 0; dim < 3; dim++) {
        if      (dr[dim] >  L[dim]/2.0) dr[dim] -= L[dim];
        else if (dr[dim] < -L[dim]/2.0) dr[dim] += L[dim];
    }

    // Calculate dr^2 instead of sqrt(dr^2), since sqrt() is a very
    // slow operation, and in this case is unnecessary
    return dr.lengthSquared();
}
```

Listing 7.4: Example implementation of `calculateDistanceSquaredBetweenAtoms` from listing 7.3.

7.3 Density

To measure the density in a uniform system consisting of just one atom type, we can use

$$\rho = \frac{Nm}{V},$$

where N is the number of atoms, m the mass of an atom, and V the volume of the whole system. But if we have a more complicated system, like in our case where we have three different atom types, liquid water in some parts of the system, and solid silica in other parts, we can not use that simple relation. What we do instead is to assiociate a volume V_i^j with each atom of type j, and calculate the density of atom type j using

$$\rho_j = \frac{m_j M}{\sum_{i=0}^{M} V_i^j},$$

where m_j is the mass an atom of type j, and M is the number of atoms of type j. We identify as the ρ_j/m_j number density. We can find the mass of an atom type from standard tables of molar masses, but we still need to find the volumes V_i^j associated with each atom. To do this we use something called Voronoi cells and the process of Voronoi tesselation. Voronoi tesselation is done by dividing the system into non-overlapping convex polyhedra (or convex polygons in 2 dimensions), with one atom in each polyhedra. The volume inside the polyhedron surrounding each atom consists of all points in space closer to that atom than any other atom.

We use the `C++`-library `Voro++` to find the Voronoi cells, and calculate the volumes of the cells. See fig. 7.3 for an illustration of a 2-dimensional Voronoi, and fig. 7.4 for a rendering of a 3D Voronoi diagram.

When measuring the Voronoi volume of each water molecule we simplify the calculations by removing all hydrogen atoms. In some systems we noticed that we had some strange vacuum bubbles in the water, very close to the silica surface in some systems. To avoid problems with this when calculating the density, we removed the upper 10% of the Voronoi volumes, to remove this long tail of high densities.

7.4 Diffusion

When we talk about diffusion in this thesis we mean the process of *self-diffusion*, which is different from "normal" diffusion, which is the net movement of a substance in the presence of a gradient, which can be for example a concentration

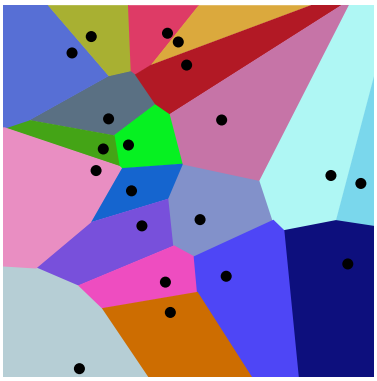


Figure 7.3: Illustration of Voronoi cells in 2 dimensions. Freely after Wikipedia Commons[48].

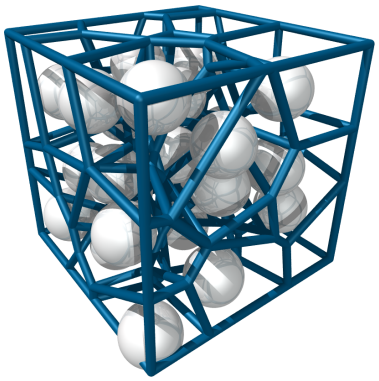


Figure 7.4: Rendering of Voronoi cells in 3 dimensions, in a system of 27 particles. Voronoi cells created using the `C++`-library `Voro++`[35, 34], and rendered using the program `povray`[32].

gradient, a temperature gradient, or a pressure gradient. With *self-diffusion* we mean the random movement in a substance that has no gradients.

Diffusion can be characterized by a constant D, which is related to the displacement of each atom relative to a initial position. We can measure this constant by measuring the mean square displacement $r_i^2(t)$ of each atom as a function of time, and average over all atoms. The mean square displacement is measured as

$$\left\langle r^2(t) \right\rangle = \frac{1}{N} \sum_{i=1}^{N} \left(\boldsymbol{r}_i(t) - \boldsymbol{r}_i(t=0) \right)^2 ,$$

where $\boldsymbol{r}_i(t=0)$ is the initial position of atom i. From theoretical considerations of the diffusion process we can relate the diffusion constant to the mean square displacement through[13, Section 4.4.1]

$$\lim_{t \to \infty} \frac{\partial}{\partial t} \left\langle r^2(t) \right\rangle = 2dD \tag{7.2}$$

where d is the spatial dimension. This means that we can find the diffusion constant in a molecular dynamics simulation by measuring the mean square displacement for many timesteps, and find the slope of this data as the diffusion constant in the limit $t \to \infty$. We are limited in that we can not actually simulate infinite number of timesteps, but have to find a reasonable number of timesteps to measure over. An example of how to sample the mean square displacement $\left\langle r^2(t) \right\rangle$ in a simulation can be seen in listing 7.5.

To measure D we see from eq. (7.2) that we have to let $t \to \infty$, but in practice we usually see that the gradient of $\left\langle r^2(t) \right\rangle$ usually have stabilzed near its final

```
double diffusionSample(System &system) {
    double rSquared = 0.0;
    for (Atom *atom : system.atoms()) {
        drVec = atom->positiom() - atom->initialPosition()
                + atom->getBoundaryCrossings()*system.size();
        rSquared += drVec.lengthSquared();
    }
    rSquared /= system.nAtoms();
    return rSquared;
}
```

Listing 7.5: An example of how to calculate the mean square displacement in a molecular dynamics simulation. Example implementation of `diffusionSample` from listing 1.9. We store the inital positions of the atoms as `atom->initialPosition()`, and when using periodic boundary conditions we count the number of times we have to translate the atom one system-size in each direction, so while the atoms will always be inside the system, the *actual* positions of the atoms can be calculated by adding `atom->getBoundaryCrossings()*system.size()` to $\boldsymbol{r}$.

value after ∼5 k timesteps of 0.050 picoseconds. We can use this to get more samples for our measurements, by using different time origos. This technique involves using different initial positions for the atoms, from different timesteps in the simulation, and then finding $\langle r^2(t) \rangle$ for $t_i \leq t \leq t_{i+n}$, where we n is the number of timesteps we want to use for each time origo. We can in theory use overlapping intervals for t, but we chose to use adjacent, non-overlapping intervals.

See fig. 7.5 for an example of how we find the diffusion constant as the gradient of $\langle r^2(t) \rangle$, using different time origos.

7.5 Tetrahedral order parameter

The tetrahedral order parameter[8] is effectively a measure of how tetrahedral a set of four points are. The tetrahedral order parameter Q for a point k is calculated as follows

$$Q_k = 1 - \frac{3}{8} \sum_{i}^{3} \sum_{j=i+1}^{4} \left[\cos \theta_{ikj} + \frac{1}{3} \right]^2, \tag{7.3}$$

where θ_{ikj} is the angle $i - k - j$ (with k in the vertex of the angle) and the two sums go over the 6 possible angles θ, between the main point and its four nearest neighbors. See fig. 7.6 for an illustration of the angles and points involved in the

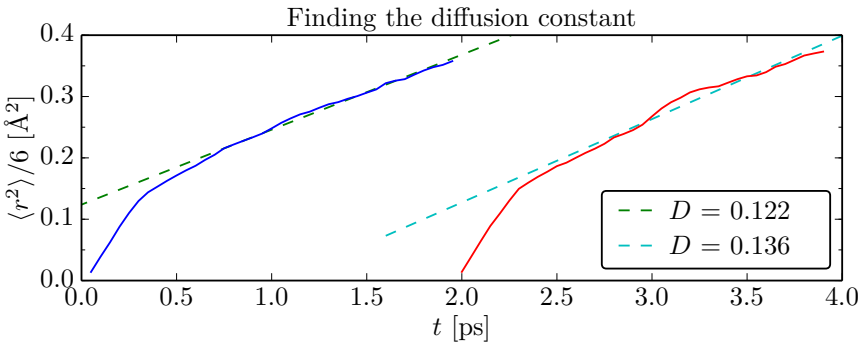


Figure 7.5: Illustration of how we find the diffusion constant, using two different time origos, with 40 timesteps of 0.050 picoseconds for each time origo. We find the diffusion constant as the gradient of $\langle r^2(t) \rangle / 6$ for the 25 last timesteps for each origo. We then use the average of the gradients for each time origo as our approximation of D.

calculation. If we have $Q = 1$ the four points are arranged in a perfect tetrahedron, with $\theta_{ijk} = 2 \arctan(2\sqrt{(2)}) \approx 109.47$ for all 6 possible angles between the four points, and as the points move away from this arrangement Q decreases following eq. (7.3) (negative Q-values are possible).

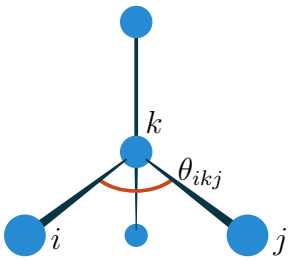


Figure 7.6: Illustration of the angles and points involved in the calculation of the tetrahedral order parameter. The blue dots are points, in our case usually water molecules. We have the center point k, and its four nearest neighbors. θ_{ikj} is the angle between point i, k and j, as indicated by the orange line.

We use the tetrahedral order parameter for investigating the coordination of water molecules relative to other water molecules. When we lower the temperature in water below freezing the average Q will increase, since the water molecules in ice has very high coordination. Liquid water also has high coordination caused by the hydrogen bonds, but much lower coordination than ice, so the distribution of Q-values are more spread out, with the mean lower than the mean for ice.

When we measure the tetrahedral order parameter we measure Q for all water molecules, defining the position of the water molecules as the position of the oxygen atom in each molecule, and plot the relative occurrence (or probability

density), $P(Q)$, to investigate the distribution of Q-values. Since Q is not dependent on several timesteps (like for example diffusion), averaging over several timesteps is trivial. Since silica is hydrophilic we expect Q to be different for water molecules near the silica surface, so we will measure $P(Q)$ as function of distance to the silica matrix.

7.6 Distance to nearest atom

To help visualize and characterize our nanoporous system we developed a program that creates a 3d map of the distance to the nearest atom, in each point in space on a regular grid. The implementation of this program is almost straighforward, but since we have to do a lot of calculations if we want to have a map with decent resolution, we have also parallellized the program to reduce the computation time.

7.7 Manhattan distance to nearest atom

Since the program from section 7.6 takes a long time to run to get decent maps with high resolution, we decided to also develop a similar program that creates a 3d map of the space, but this time creating a map with the Manhattan distance from each point on the grid to the nearest atom. To ease calculation we this time used a method inspired by the voxelation technique from section 7.2.

This program first divides the system into $n_x \times n_y \times n_z$ voxels, and make a 3d matrix of the same size for storing the Manhattan distance to the nearest atom in each point. We first give all voxels with one or more atoms in them the distance 0. We then label the rest of the voxels using an iterative method, increasing the number by one for each iteration. In each iteration we find the voxels that have a neighbor voxel labelled with the previous label (`label-1`), using 6-nearest-neighbor connectivity, and give them the current label. When all voxels are labelled, they should have a label corresponding to the Manhattan distance to the nearest atom.

Although the Manhattan distance is not as useful as the regular Euclidean distance we calculate using the "distance to atom"-program, the benefit is that making a 3d map of the Manhattan distances uses about 3% of the time that "distance to atom" uses for the same system and same resolution. For a system of 347k atoms, a nanoporous silica system, using 256 voxels in each direction (a total of $256^3 \approx 16.7$M voxels), the program that finds the Manhattan distance

uses about 5 seconds, but the program that finds the Euclidean distance uses 2 minutes and 27 seconds.

Chapter 8

Studied systems

We have do experiments on a total of 8 different systems, all consisting of a slab of silica with a fracture in the center filled with water. Four of them are "reference" systems with just a flat fracture in the center, and the other four are systems with a random fracture with different geometries.

All systems are created using the experimental procedure from section 6.1. The systems are initialized as a perfect crystal of β-cristobalite. The system is then brought to 4500 Kelvin using a thermostat, to melt the silica crystal. It is then cooled back down to 300 Kelvin, and a fracture is cut out of the solid slab of silica, the system dangling ends are passivated, and the fracture is filled with water. The system is then thermalized at 300 Kelvin.

A summary of the different systems can be seen in table 8.1, where we have listed the dimension, porosity, the number of atoms, and the number of SiO_2 and water species in each system.

In all systems with narrow pores and fractures we noticed that the water filling method had some problems. This is caused by the voxelation method used to fill the system, where we divide the system into voxels, mark all voxels with atoms in them (before filling the system with water) as occupied, and then put one water molecule in each unoccupied voxel. When marking occupied voxels we usually end up marking a lot of the voxels at the silica surface as occupied. This means that when we have very narrow pores, with the distance between the pore walls in the same range as the voxel size, a large fraction of the pore will be occupied voxels, and the resulting water density in the pore will be lower than the expected density. To rectify this we use higher input densities when filling narrow fractures and pores with water.

System	*Dimensions* [Å]	ϕ [%]	r [Å]	N	N_{SiO_2}	N_{H_2O}
Rough fracture #1	$179 \times 179 \times 179$	∼12	-	393 k	111 k	19 k
Rough fracture #2	$172 \times 172 \times 172$	∼13	-	347 k	97 k	18 k
Rough fracture #3	$172 \times 172 \times 172$	∼13	14.4	349 k	99 k	16 k
Rough fracture #4	$172 \times 172 \times 172$	∼23	28.8	368 k	89 k	34 k
Reference #1	$179 \times 179 \times 179$	48	86	260 k	25 k	60 k
Reference #2	$179 \times 179 \times 179$	48	86	271 k	25 k	64 k
Reference #3	$143 \times 143 \times 57$	25	14.4	90 k	19 k	10 k
Reference #4	$143 \times 143 \times 57$	50	28.8	107 k	13 k	22 k

Table 8.1: An overview of the 8 different systems we have done experiments on. "Flat fracture" 1 through 4 are reference systems, that consist of a silica slab with a single flat fracture filled with water. "Rough fracture" 1 through 4 consist of a silica slab with different water-filled fractures with different geometries.
ϕ is the approximate porosity of the system, defined as the volume of the fracture relative to the volume of the whole system. r is the distance between the surfaces used to create the fracture. N is the total number of atoms (silicon, oxygen and hydrogen), N_{SiO_2} is the number of SiO_2-units, and N_{H_2O} is the number of water molecules.

Rough fractures

The fractures in the systems labelled "rough fracture" are all created using periodic surfaces with a Hurst exponent close to 0.75, generated using successive random additions from section 5.2.2, and the fractures cut out using the method from section 5.3. "Rough fracture" #1 and #2 are created using different surfaces for the top and bottom of the fracture, while "rough fracture" #3 and #4 are created using the same surface repeated twice for the top and bottom of the fracture, with a distance of respectively 14.4 and 28.8 Å between the surfaces, which gives an approximately constant width fracture.

When filling system #1 and #2 with water we used an input density of 1050 kg/m^3, and for system #3 and #4 we used 1273 kg/m^3.

Reference systems

To compare with the fracture systems we have also prepared four "reference" systems, all consisting of one flat pore with constant width.

"Reference" #1 and #2 both have a 86 Å wide pore. When filling system #1 and #2 with water we used an input density of respectively 1050 and 1126 kg/m^3.

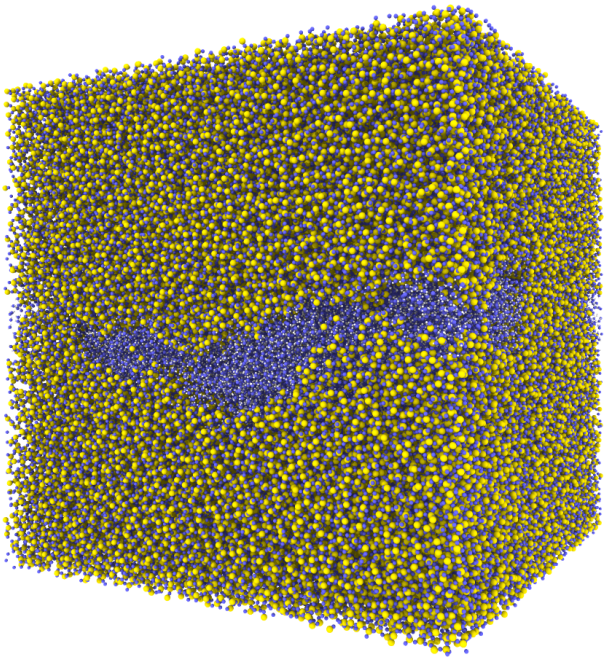
These two systems will serve as good references for the behaviour and structure of water near the silica surface, and in bulk-like conditions, as we expect the water in the middle of the pore to display bulk-like behaviour.

"Reference" #3 and #4 consist of flat pores that are respectively 14.4 and 28.8 Å wide, which we will compare to the random fracture systems with random uniform fractures in them. When filling the pores in these systems with water we used input densities of 1273 kg/m^3. This is a pretty high density, but the voxelation method had some problems in very narrow fractures, giving a lower density than intended, which made us use a higher density to reach an approximate density of 1000 kg/m^3.

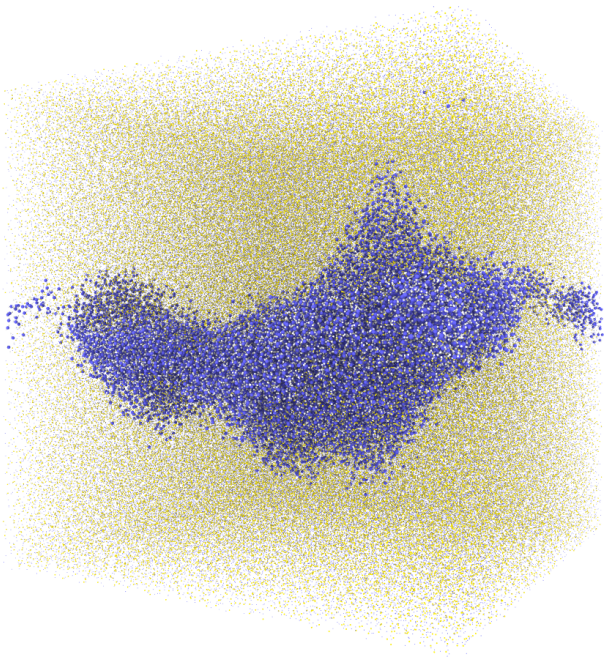
8.1 Visualizations

We have made some renderings of the systems we have studied, as can be seen in figs. 8.1 to 8.6. All renderings and visualizations were made using the program Ovito[40], using the built-in open-source "Tachyon" rendering engine.

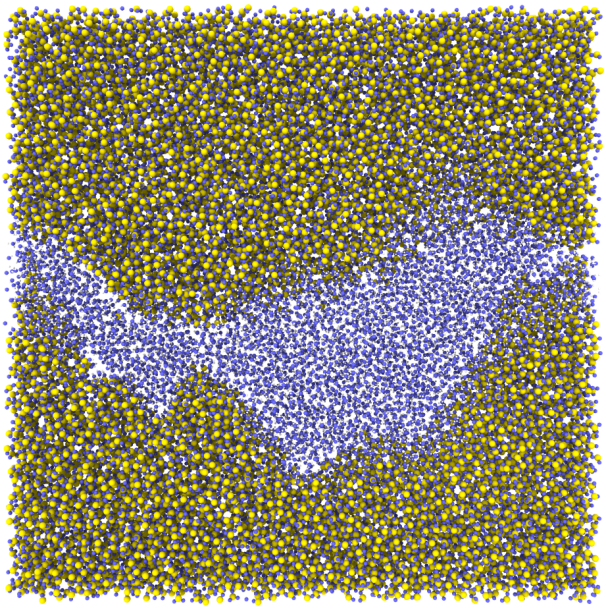
In the renderings in this section we have colored the silicon atoms yellow, the oxygen atoms blue, and the hydrogen atoms white. The silicon atoms have been given a radius of 1 Å, the oxygen atoms 0.6 Å, and the hydrogen atoms 0.3 Å.



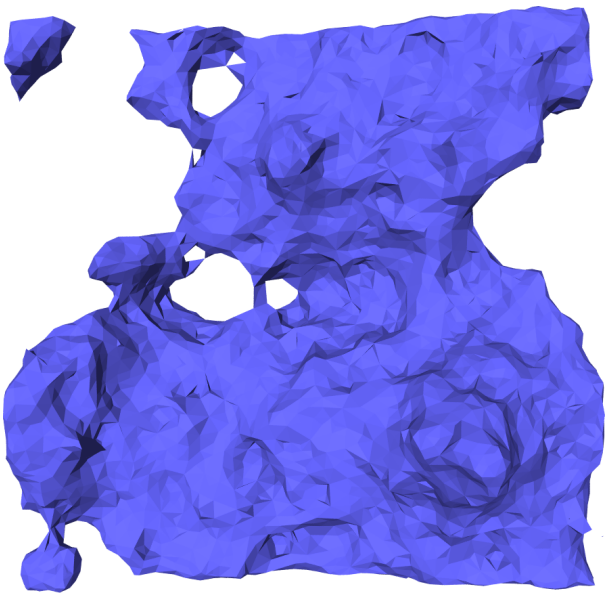
(a) The whole system.



(b) The whole system, with the size of the silicon and silica-oxygen atoms reduced to 0.1 Å.

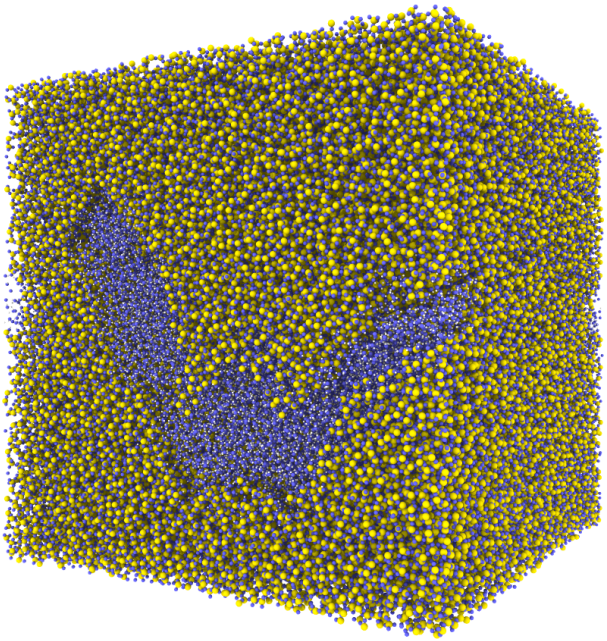


(c) 20 Å thick slice.

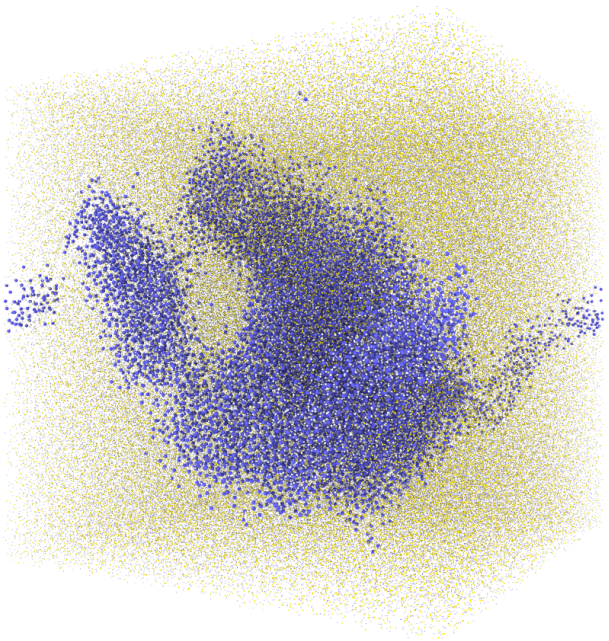


(d) The pore volume.

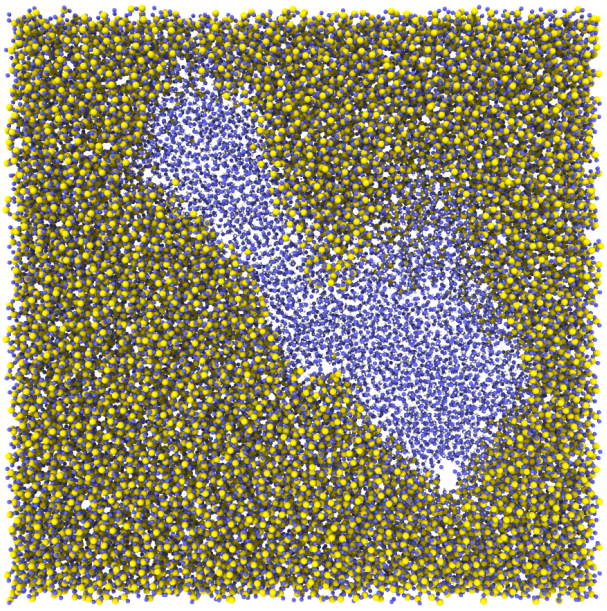
Figure 8.1: "Rough fracture #1", a randomly generated fracture with varying width. Generated from two random surfaces. The size of this system is $179 \times 179 \times 179$.



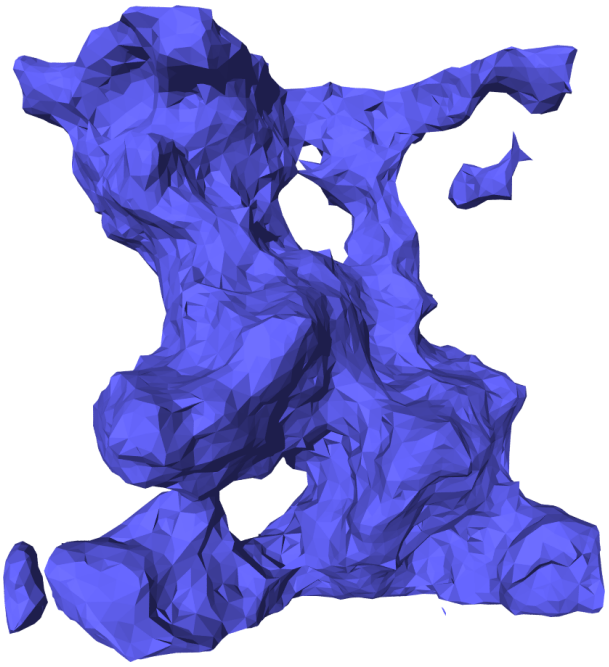
(a) The whole system.



(b) The whole system, with the size of the silicon and silica-oxygen atoms reduced to 0.1 Å.

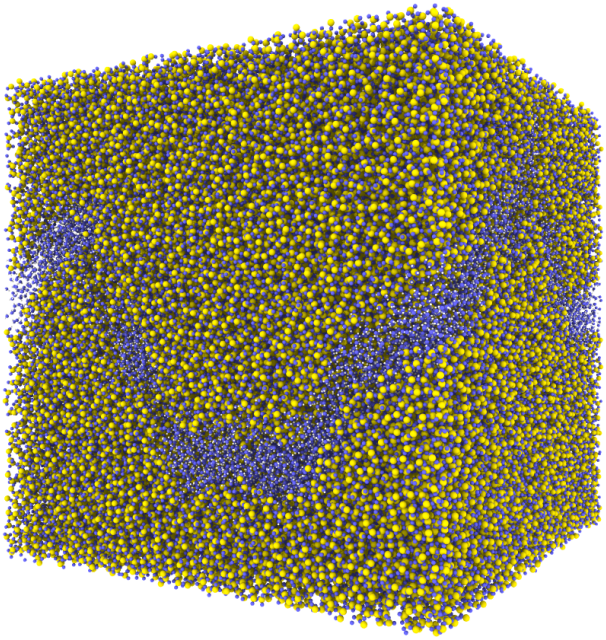


(c) 20 Å thick slice.

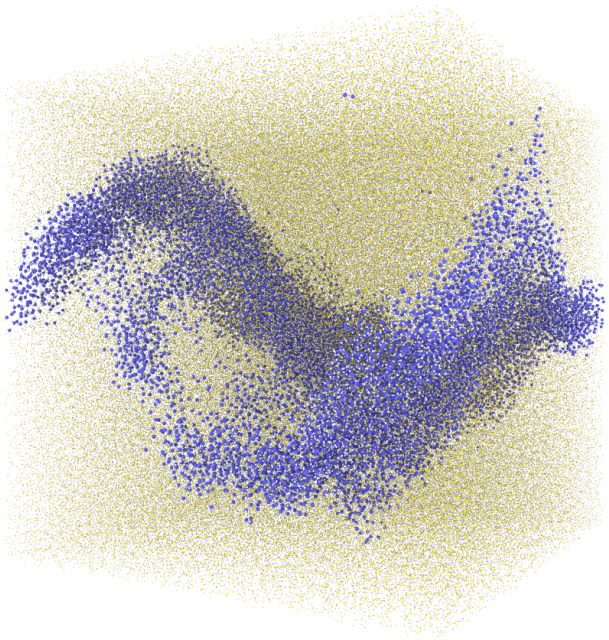


(d) The pore volume.

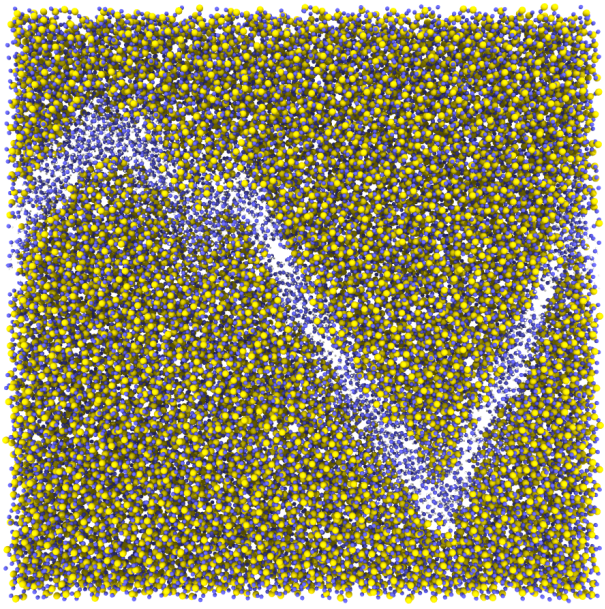
Figure 8.2: "Rough fracture #2", a randomly generated fracture with varying width. Generated from two random surfaces. The size of this system is $172 \times 172 \times 172$.



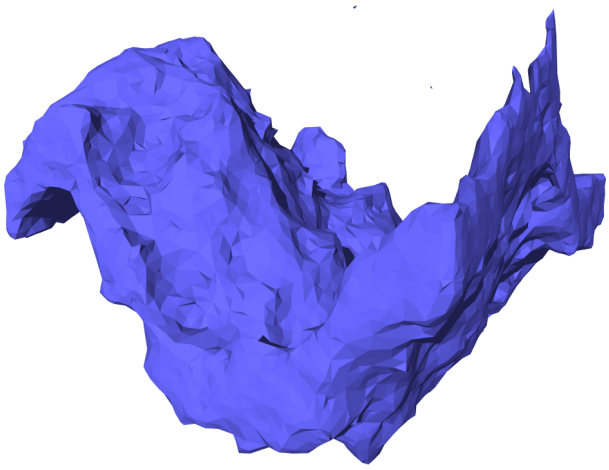
(a) The whole system.



(b) The whole system, with the size of the silicon and silica-oxygen atoms reduced to 0.1 Å.

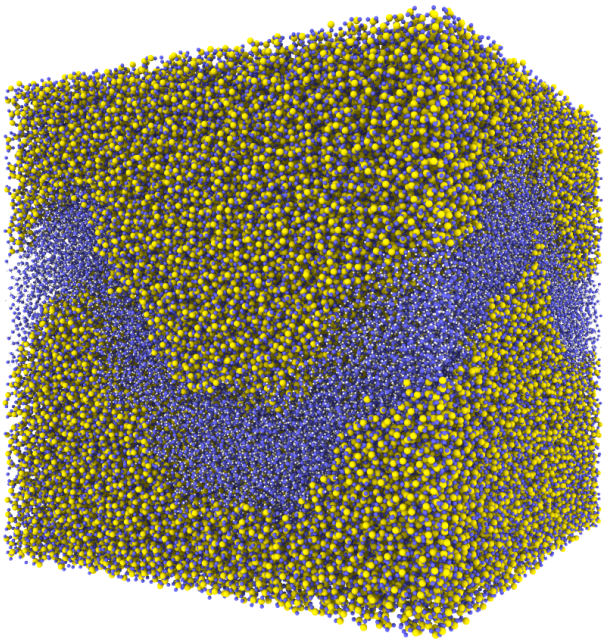


(c) 20 Å thick slice.

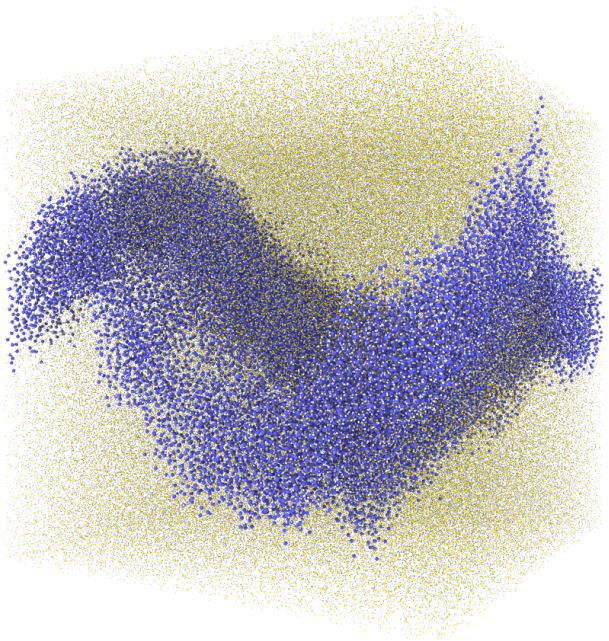


(d) The pore volume.

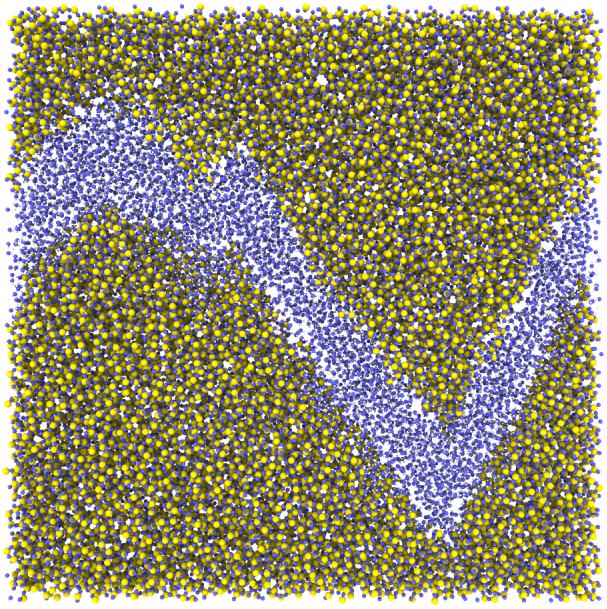
Figure 8.3: "Rough fracture #3", a randomly generated fracture generated from one surface repeated for the top and bottom half, with 14.4 Å between the surfaces, giving approximately uniform width of the pore. The size of this system is $172 \times 172 \times 172$.



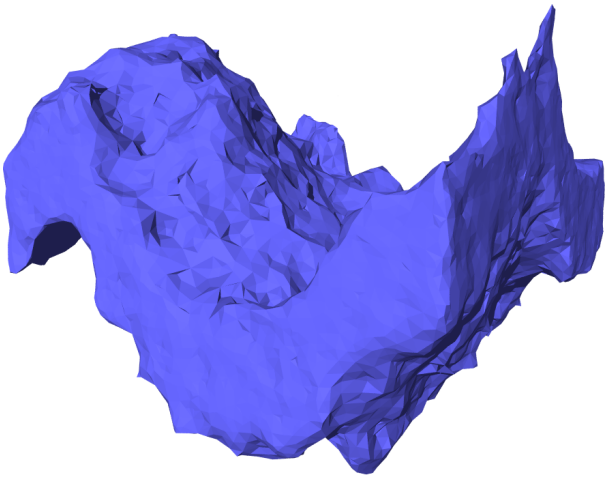
(a) The whole system.



(b) The whole system, with the size of the silicon and silica-oxygen atoms reduced to 0.1 Å.

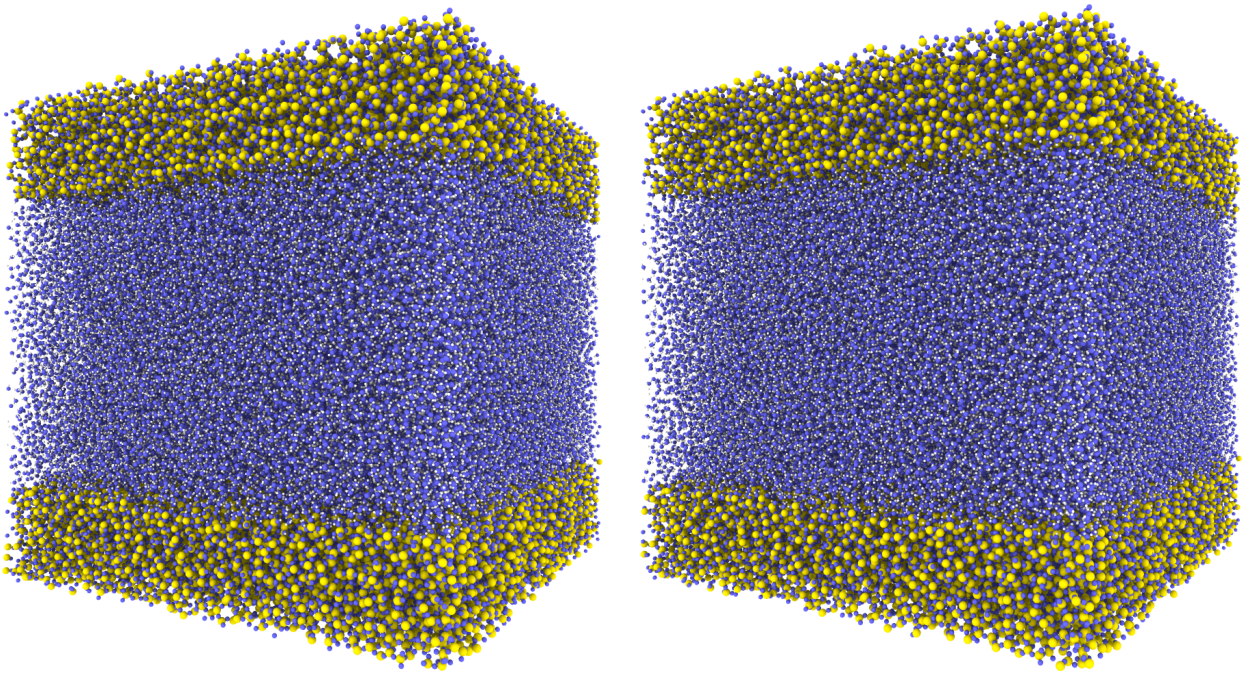


(c) 20 Å thick slice.



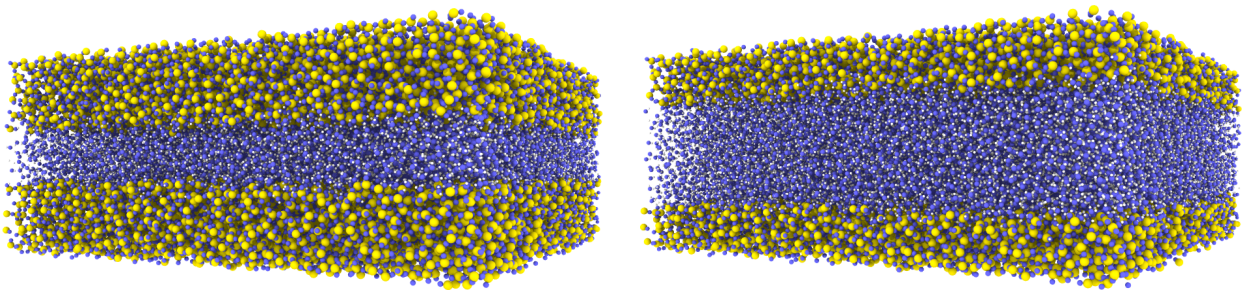
(d) The pore volume.

Figure 8.4: "Rough fracture #4", a randomly generated fracture generated from one surface repeated for the top and bottom half, with 28.8 Å between the surfaces, giving approximately uniform width of the pore. The size of this system is $172 \times 172 \times 172$.



(a) "Reference #1". **(b)** "Reference #2".

Figure 8.5: Reference systems #1 and #2, 86 Å wide flat pores. Both these systems have size $179 \times 179 \times 179$.



(a) "Reference #3". **(b)** "Reference #4".

Figure 8.6: Reference systems #3 and #4, respective a 14.4 and a 28.8 Å wide flat pore. Both these systems have size $143 \times 143 \times 57$.

Chapter 9

Results

Here we present the result of all the measurements we have done. Most of the measurements have been done for 200 different states for each system, with 100 timesteps of 0.050 picoseconds between each state (5 picoseconds between each state). Most measurements have been measured as function of distance to the silica matrix to study the effects of the interactions between water and silia, and we have tried to find the bulk-like behaviour of all measurements for comparison.

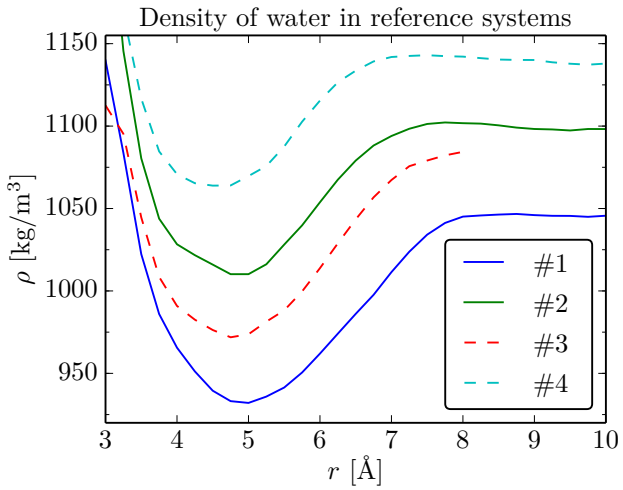
We will compare results between the random rough fractures to the results for bulk like-water in reference system #1 and #2, and against the two narrow flat reference systems #3 and #4, to see if the changes from flat pores to rough fractures have an effect on the water. The results for the rough fracture systems will also be compared against each other to see if the different fracture geometries have any effect.

We will first present and analyze the results from each measurement individually in sections 9.1 to 9.5, and then later in chapter 10 we will give a summary of the results, and compare the results from the different measurements against each other to see if there are any further conclusions to be drawn.

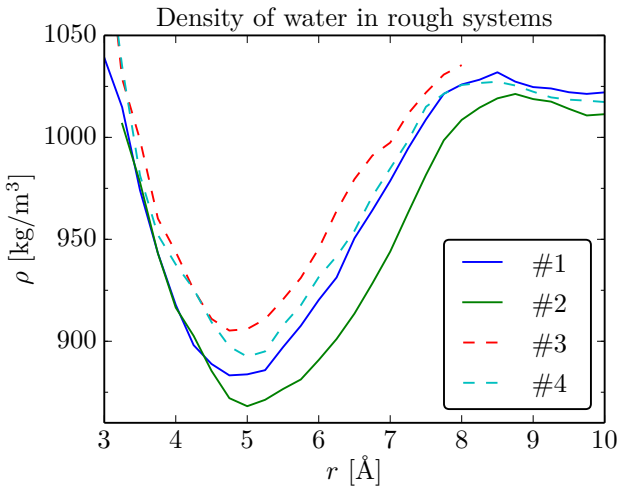
9.1 Density of water

We have measured the density of water ρ as function of distance to the silica matrix in all of our systems, averaged over 200 states for each system, with 100 timesteps of 0.050 picoseconds between each state (5 picoseconds between each state). The results are plotted in figs. 9.1 to 9.2. We measured for distances ranging from 0 to 10 Å from the silica matrix, in steps of 0.25 Å. We have also

measured the bulk density in the systems where this was possible, the results of which are listed in table 9.1. The plots start at 3.0 Å, since water molecules closer to the silica matrix than this are most likely bound to silica atoms, and are part of the passivating silanol groups, as we saw in section 7.1.1.



(a) Dashed lines are 14.4 Å (red, #3) and 28.8 Å (teal, #4) narrow flat pores, solid lines are 86 Å wide flat pores.



(b) Dashed lines are 14.4 Å (red, #3) and 28.8 Å (teal, #4) narrow fractures, solid lines are random fractures.

Figure 9.1: Water density (ρ) as function of distance to silica matrix (r) in **(a)** all four reference systems (flat pores) and **(b)** rough fracture systems.

The most significant trend we notice in fig. 9.1, for all systems, is that the water density seems stable at 9-10 Å from the silica matrix, but as we move closer than this we see a clear reduction in density. When we go from 10 Å and move closer to the matrix we see that the density falls off at around 7-8 Å and is reduced by almost 10% at around 5 Å. The density then increases to densities higher than the initial densities (the ones at 10 Å) when we move towards 3 Å. The relative reduction and then increase in density seems to be similar in all systems with similar overall densities, but in the systems with the highest overall density the relative change seems to be a bit smaller than in the other systems.

We now look at the density in the four reference systems, which is plotted in fig. 9.1a. We see that the densities all have the same quantitative behaviour as we move further from the silica matrix, even though the net density is very different in the four systems, with the densities stabilizing at values ranging from 1050 to almost 1150 kg/m^3 at 8-10 Å. One trend we notice is that the point where the density stabilizes, at around 7 Å, seems to appear closer to the matrix when we increase the overall density (see also the dashed plots and arrows in fig. 9.2a

for a visualization of this). The minimum point seems to appear at approximately the same distance for the systems with a 86 Å wide flat pore (system #1 and #2), but a bit closer to the matrix for the narrow flat pores (system #4 and #3).

In fig. 9.1b we have plotted the density in the four random fracture systems. We see that the density is very similar in all systems, at all distances to the matrix, and that the density seems to stabilize between 1010 and 1025 kg/m^3 at 8-10 Å from the silica matrix for all systems. We see that even though these four fractures have different geometries (especially system #1 and #2 are very different from system #3 and #4), the behaviour of the density as we go from 8 to 3 Å from the silica matrix seems to be the same in all four systems. We also see that the density seems peak at around 8 Å, and stabilize at a value a bit lower than this peak as we go further from the silica matrix. This peak is similar to a peak observed around 5 Å from the silica matrix by Bonnaud et al. in [5] (see Figure 6 in the article). In the article they use a different measure for the distance to the silica matrix, which might explain the difference in the distance at which the peak is observed.

We have estimated the bulk water density by the same technique we use for measuring water density as function of distance to the silica matrix, but now by averaging the density for all water-oxygen atoms further away from the silica matrix than a certain distance (typically 10 Å or more). This measurement requires that we have a fracture at least twice as wide as this distance to have any atoms we can measure the density of, so estimating the bulk density was only possible in reference systems #1 and #2 with a flat, constant fracture of 86 Å, and in the two random fractured systems #1 and #2, which have fractures with varying width.

	$r > 10$ Å		$r > 30$ Å	
System	ρ [kg/m^3]	N	ρ [kg/m^3]	N
Reference #1	1038.4	49k	1038.5	20k
Reference #2	1092.7	52k	1090.4	21k
Rough #1	1017.9	5.0k	-	0
Rough #2	1002.8	4.2k	-	0

Table 9.1: Estimated bulk water densities, and the number of water molecules used in the calculations. Estimated using Voronoi tesselation, averaged over voronoi volumes for all water-oxygen atoms further away from silica matrix than 10 and 30 Å (hydrogen atoms were removed before Voronoi tesselation).

The estimated bulk densities are listed in table 9.1, where we have measured

the bulk density for water atoms at least 10 Å and at least 30 Å from the silica matrix. We see that the estimated bulk density does not change if we use 10 or 30 Å as the minimum distance from the matrix, indicating that a limit of 10 Å is adequate, and that the density does not change much from 10 to 30 Å. If we compare the bulk densities to the plots of the density as function of distance from the matrix in figs. 9.1a and 9.1b, we see that the density seems to be close to the bulk density at 10 Å, indicating that water have close to bulk-like properties from around 8 Å and further from the silica matrix.

In fig. 9.2a we have plotted the density in all eight systems we have simulated. For easier comparisons we have also tried normalizing the density against an estimated bulk density (estimated from the density at 8-10 Å), plotted in fig. 9.2b. In both figures we see the same trend as before, where the falloff for the density seems to move closer to the matrix as we increase the overall density. We see a hint of a similar trend for the minimum point of the density, but not nearly as clear as for the falloff point. We also notice that the falloff seems to be closer to the silica matrix for the rough fracture systems than for the flat pores, with the falloff being near 8 Å in the rough fracture systems, but from 7 to 8 Å for the flat pores, depending on the overall density.

Filling density vs. actual density

To check the accuracy of the method we use for filling the fractures and pores with water, we want to compare the input densities used when filling the pores to the resulting density in the systems. We have already found the bulk density in four of the systems, but in the other systems we are not able to measure the density in the same way. What we do in these systems ("rough fracture" #3 and #4, and reference system #3 and #4) is to approximate the bulk density by looking at the value of the density near 10 Å from the silica matrix. This should give us a rough estimate of the bulk density in the system[1].

The results can be seen in table 9.2. We see that using density at 10 Å gives a good approximation to the measured bulk density, by comparing the bulk density measured for atoms further away than 10 Å to the density estimated from the density at 10 Å, at least for the system where we are able to measure the bulk density. We see that the method for filling the fractures and pores with water performs well when the pores and fracures are very large. But we also see that in the systems with very narrow pores, the method for filling the pore with water seems to struggle with acheving the wanted density. This is cause by the

[1]In rough fracture system #3 and reference system #3 we do not have any measurements for water molecules around 10 Å from the matrix, since these systems have pores that are narrower than 20 Å. For these systems we extrapolate the the density at 10 Å from the values we have.

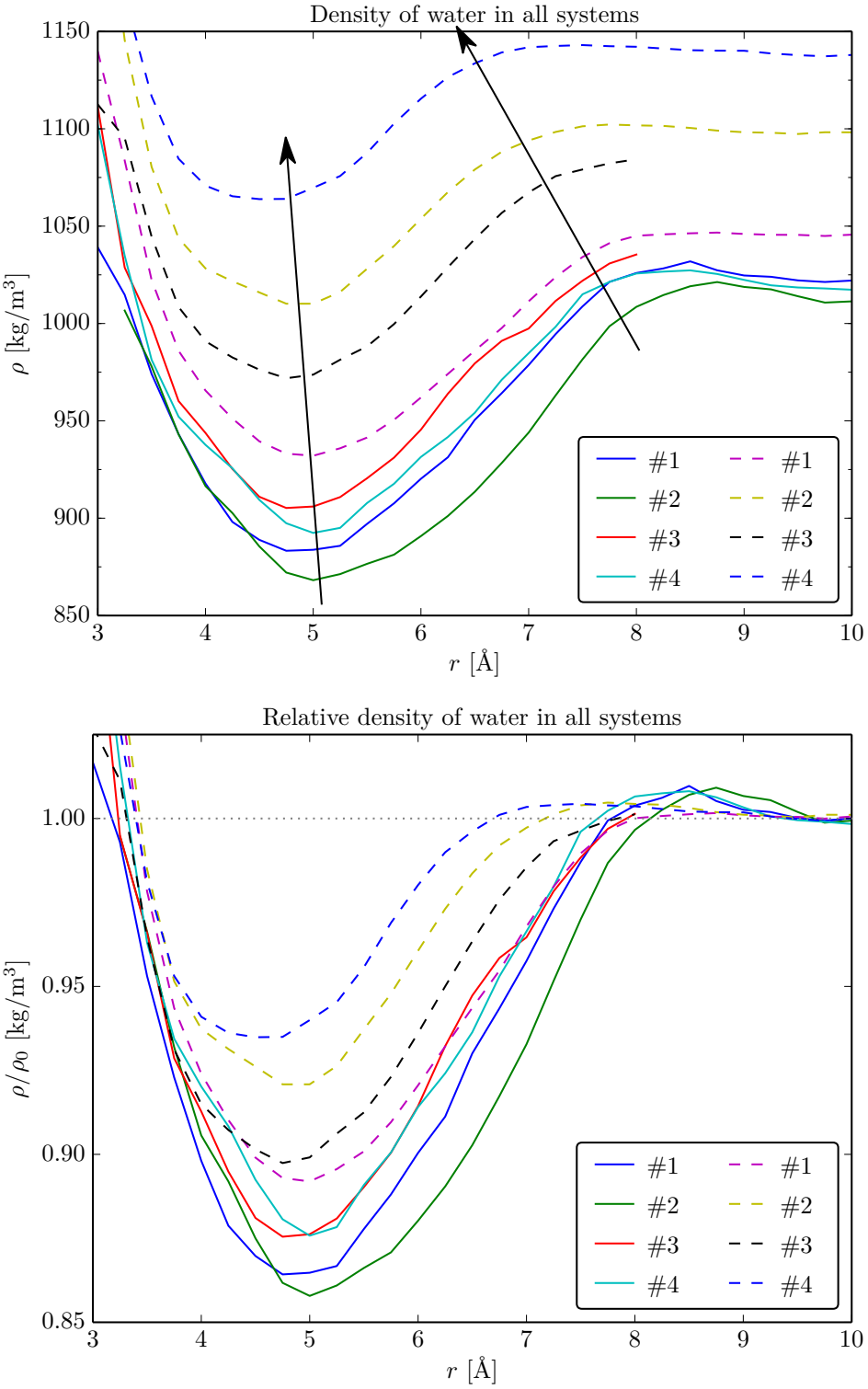


Figure 9.2: Density of water in all systems, as function of distance from the silica matrix. Dashed lines are reference systems, and solid lines rough fracture systems. In **(b)** the density has been normalized against the approximate bulk density, estimated from the density at 10 Å from the silica matrix (see "Approx ρ at 10 Å" in table 9.2 for these normalization factors).

System	Input ρ [kg/m^3]	Measured ρ [kg/m^3]	Approx. ρ at 10 Å
Reference #1	1050	1038	1045
Reference #2	1126	1093	1098
Reference #3	1273	-	1085
Reference #4	1273	-	1135
Rough #1	1050	1018	1022
Rough #2	1050	1003	1010
Rough #3	1273	-	1025
Rough #4	1273	-	1016

Table 9.2: A comparison of the estimated and measured bulk water density in the systems we have simulated, with the input density we used when filling the fractures with water. "Approx ρ" is the density we see at 10 Å from the silica matrix, and "Measured ρ" is the average density for atoms further away from the silica matrix than 10 Å.

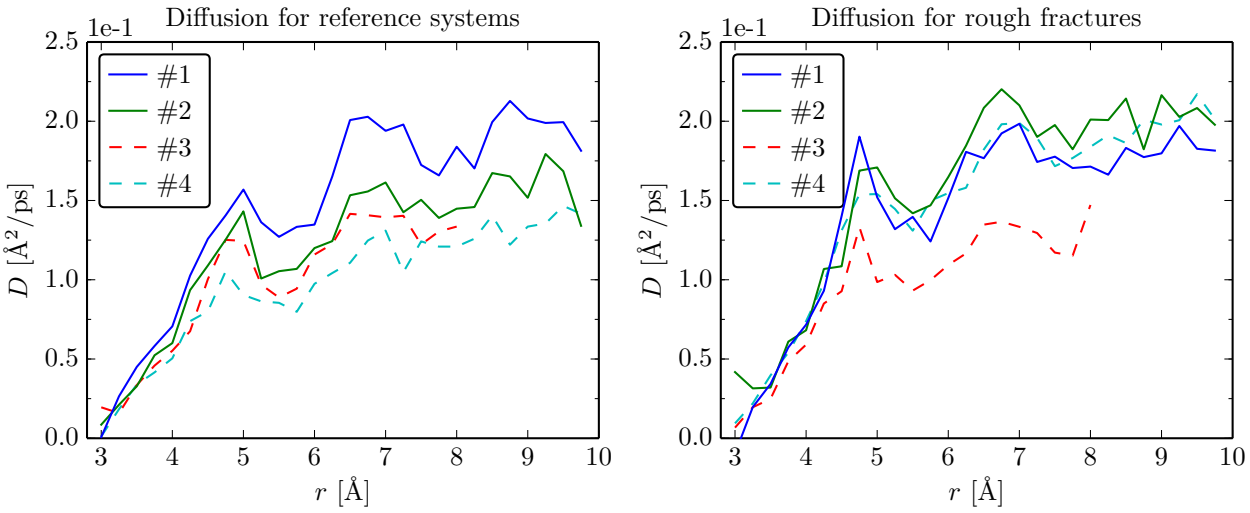
voxelation method we use when finding room for water molecules, as expected and noted previously.

9.2 Diffusion

We have measured the diffusion constant D for water in the four reference systems, and the four random fracture systems. We have measured D as function of distance to the silica matrix, in steps of 0.25 Åfor the distance. We used 200 states for each system, with 100 timesteps of 0.050 picoseconds between each state (5 picoseconds between each state). To improve the statistics we divided each set of states into 5 non-overlapping origos, with 40 states per origo. The results are plotted in figs. 9.3 to 9.4. We have also measured the bulk diffusion constant, the results of which are listed in table 9.3.

The overall trend we see in all plots and for all systems is that the diffusion constant is reasonably stable from 10 down to 6-7 Å from the silica matrix, where it does a dip between 5 and 6 Å, goes over a peak near 5 Å, and then goes almost linearly from this peak to zero at 3 Å. We see that the diffusion constant for all systems follow this trend, even though the overall diffusion is different in different systems.

We now look at the diffusion for the four reference systems, which are plotted in fig. 9.3a. We see that system #1 has overall higher diffusion than system #2, even though these two systems have very similar characteristics, as both consist of a 86 Å wide flat pore. We notice a somewhat higher diffusion in the 14.4 Å



(a) Dashed lines are respectively a 14.4 Å (red, #3) and a 28.8 Å (teal, #4) flat pore, and solid lines are 86 Å wide flat pores.

(b) Dashed lines are respectively a 14.4 Å (red, #3) and a 28.8 Å (teal, #4) narrow rough fracture, and solid lines are random rough fractures.

Figure 9.3: Diffusion constant D as function of the distance from the silica matrix r. In **(a)** we have all reference systems with flat pores, and in **(b)** all rough random fractures.

flat pore in system #3 than in the 28.8 Å flat pore in system #4. The diffusion constant behaves very similarly in all four systems.

In fig. 9.3b we have the diffusion for the four random fracture systems. We see that the diffusion constant behaves very similarly in all four systems, but that in system #3, which is the 14.4 Å narrow fracture, we have overall lower diffusion.

The bulk diffusion constants have been estimated in the two reference systems with 86 Å wide flat pores (reference #1 and #2), and in rough system #1 and #2, by measuring D for all water molecules further away from the silica matrix than 10 Å. The results can be seen in table 9.3. The other four systems have very narrow pores and fractures, with most of the water molecules closer to the silica matrix than 10 Å, and as we see from fig. 9.3 the transport properties of water in these regions are different from bulk water, so we have not been able to estimate the diffusion of bulk-like water in these systems.

If we compare the bulk diffusion constants in table 9.3 with the plots in fig. 9.3, we see that the diffusion constant has reached values close to the bulk value at around 6-7 Å from the silica matrix.

In fig. 9.4a we have plotted diffusion for the two random fractures with varying

System	Bulk D [Å^2/ps]	N
Reference #1	0.202	48k
Reference #2	0.179	50k
Rough #1	0.198	4.3k
Rough #2	0.209	3.4k

Table 9.3: Bulk diffusion constant for water (for water molecules more than 10 Å from the silica matrix), and the number of water molecules used in the calculations (N).

pore width (the two solid lines) together with all four reference systems (the four dashed lines). We see that the behaviour of D in both the rough systems are very close to the behaviour in reference system #1, which is the 86 Å wide flat pore, and that the overall diffusion is lower in the three other reference system than in the plotted rough fracture systems.

In fig. 9.4b we have plotted the diffusion for the two random narrow fractures with uniform width (the two solid lines) together with all four reference systems (the four dashed lines). We see that the diffusion in the 14.4 Å narrow fracture ("Rough #3") matches the the diffusion in reference system #3 very well, which is a 14.4 Å flat pore. We also notice that the 28.8 Å narrow fracture has overall higher diffusion than reference system #4, which has a 28.8 Å flat pore, but that it matches reference system #1 very well, which has a 86 Å flat pore.

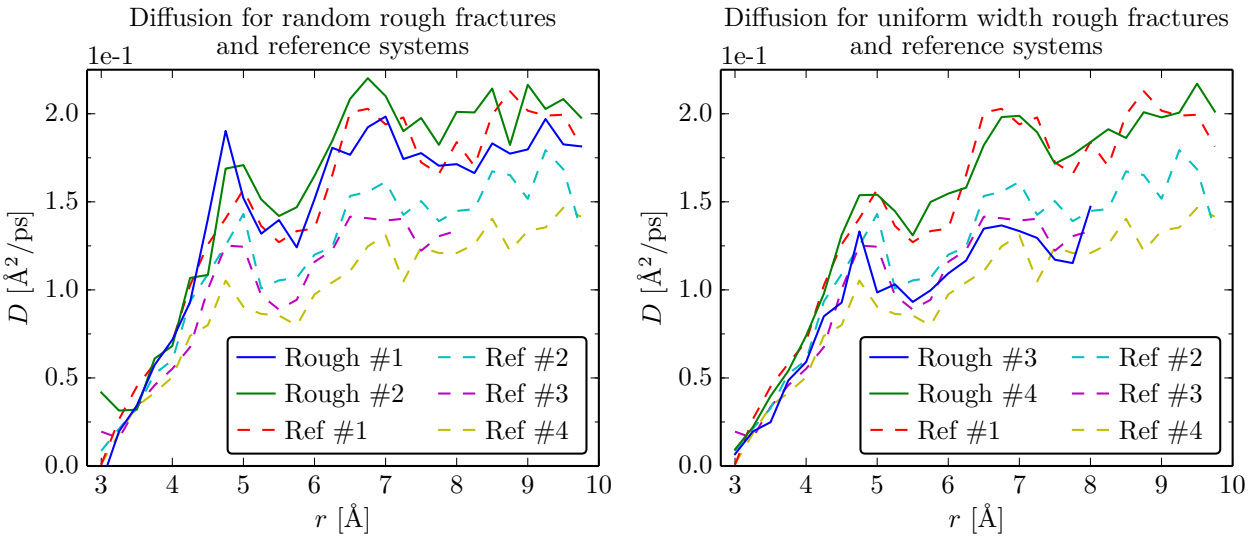


Figure 9.4: Diffusion constant (D) as function of distance from silica matrix (r). Solid lines are **(a)** rough fracture system #1 and #2 (normal rough fractures), and **(b)** the narrow uniform width fractures (fracture system #3 and #4). Dashed lines are the four reference systems, in both **(a)** and **(b)**.

9.3 Tetrahedral order parameter

We have measured the thetrahedral order parameter for our four reference systems, and the four random fracture systems, and plotted the relative occurrence (or probability) $P(Q)$, for water molecules with different distances from the silica matrix. We used used 150 bins for the Q-values, with $\Delta Q = 0.01$, and bins of $\Delta r = 0.5$ for the different distances to the matrix, for all measurements presented here. The results are plotted in figs. 9.5 to 9.9. We have also measured the bulk order parameter, which is plotted in fig. 9.5.

We have estimated $P(Q)$ in bulk water by measuring the tetrahedral order parameter for all water molecules further from the silica matrix than 10 Å, in the two reference systems with 86 Å wide flat pores. The results for bulk water can be seen in fig. 9.5. We see that we get two peaks in the distribution, one just above $Q = 0.5$ and one right below $Q = 0.75$. These results, and the two peaks, fit well with the results of Kumar et al. in [20] for water at 300 K.

In fig. 9.6 we have plots of $P(Q)$ for all four reference systems, and in fig. 9.7 we have plots for all four random rough fracture systems. The overall trend we see is that the different geometries and pore widths does not affect $P(Q)$ a lot, but we see a clear change in the structure of water as we get closer than 5.5 Å to the silica matrix. From 6.5 to 5.5 Å $P(Q)$ is very similar to bulk, but as we go closer

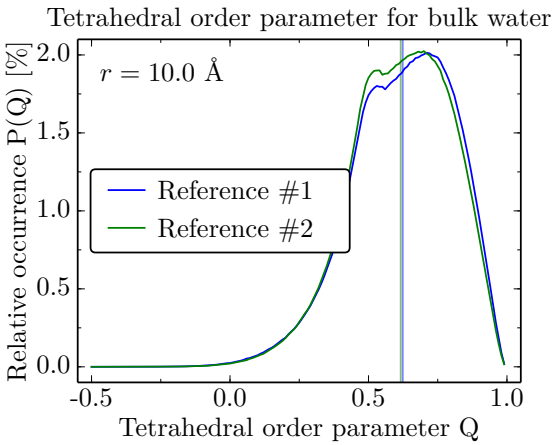


Figure 9.5: Plot of tetrahedral order parameter for bulk water (water molecules further than 10 Å from the silica matrix), in the two reference systems with 86 Å wide flat pores (reference system #1 and #2). The average Q is indicated by a vertical line.

than this the right peak get gradually lower, and $P(Q)$ is more spread out. This indicates a clear change in the structure of water molecules when we get close to the silica matrix, a change that seems almost independent of the structure of the surface of the matrix, and the width of the fracture.

We now look at fig. 9.6, where we have plotted $P(Q)$ for all four reference systems. We see that $P(Q)$ is almost identical for all four reference systems, at all distances from the silica matrix.

In fig. 9.7 we have plotted $P(Q)$ for all four rough fracture systems, with random fractures in solid lines and uniform width fractures in dashed lines. We see that all four systems have very similar $P(Q)$ from 3.0 to 5.0 Å, but at 5.5 Å and further out the right peak near $Q = 0.75$ peak seems to rise faster for the two narrow fractures of uniform width (#3 and #4, the two dashed lines) than for the fractures with random width. At 6.0 and 6.5 Å the right peak for system #1 seems to start catching up to the uniform width fractures.

In fig. 9.8 we have plots of $P(Q)$ for the two random rough fractures with varying pore width (the solid lines), and for all four reference systems (the dashed lines). We see that the two peaks in $P(Q)$ generally lie lower for the rough fractures than the reference systems, for distances smaller than 5.5 Å, but that this difference disappears at 6.0 and 6.5 Å, where they all have bulk-like appearance. There are no clear differences between the two rough fracture systems, other than a slightly higher right peak for rough fracture system #1 at 6.5 Å.

In fig. 9.9 we have plots of $P(Q)$ for the two random fractures with uniform width (the solid lines), and for all four reference systems (the dashed lines). We again

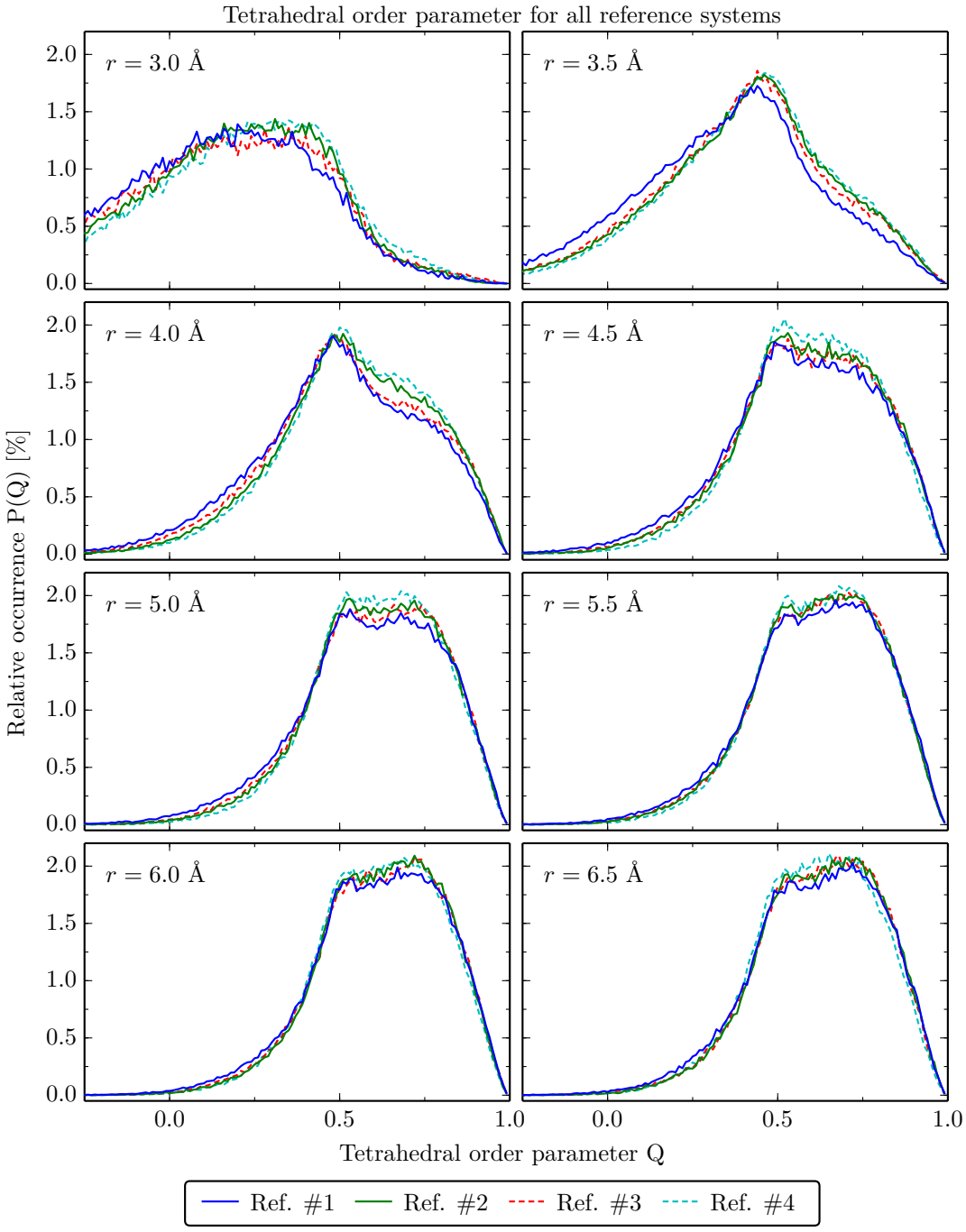


Figure 9.6: Plots of $P(Q)$ for all reference systems, for different distances to the silica matrix r. The solid lines are 86 Å wide flat pores, and the dashed lines respectively a 14.4 Å wide flat pore (#3, red) and a 28.8 Å wide flat pore (#4, teal).

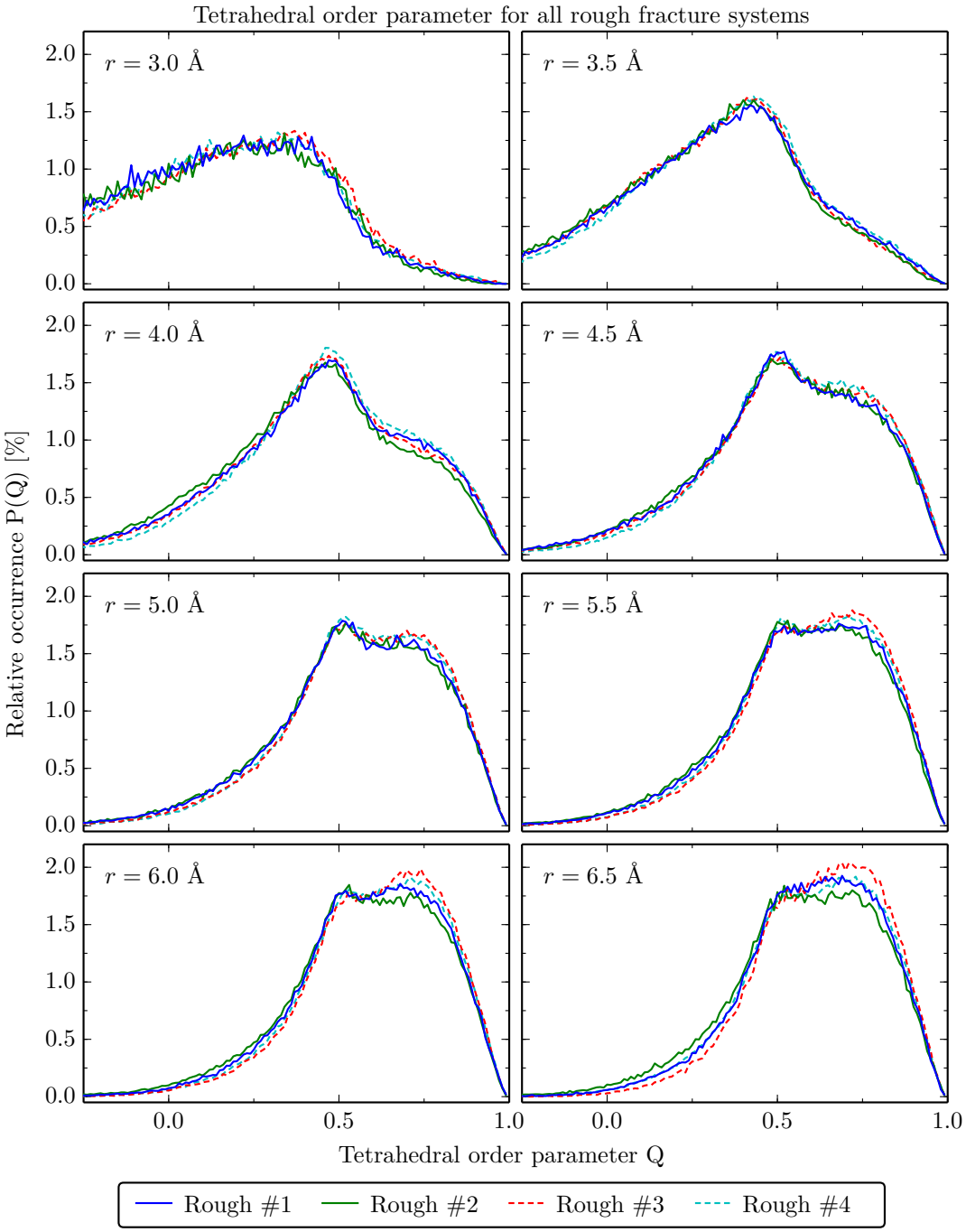


Figure 9.7: Plots of $P(Q)$ for all all rough fracture systems, for different distances to the silica matrix r. The solid lines are random fractures generated from two different surfaces, and the dashed lines are random fractures of uniform width, generated from the same random surface repeated for the top and bottom of the fracture, with a distance of respectively 14.4 Å (red, #3) and 28.8 Å (teal, #4) between the surfaces.

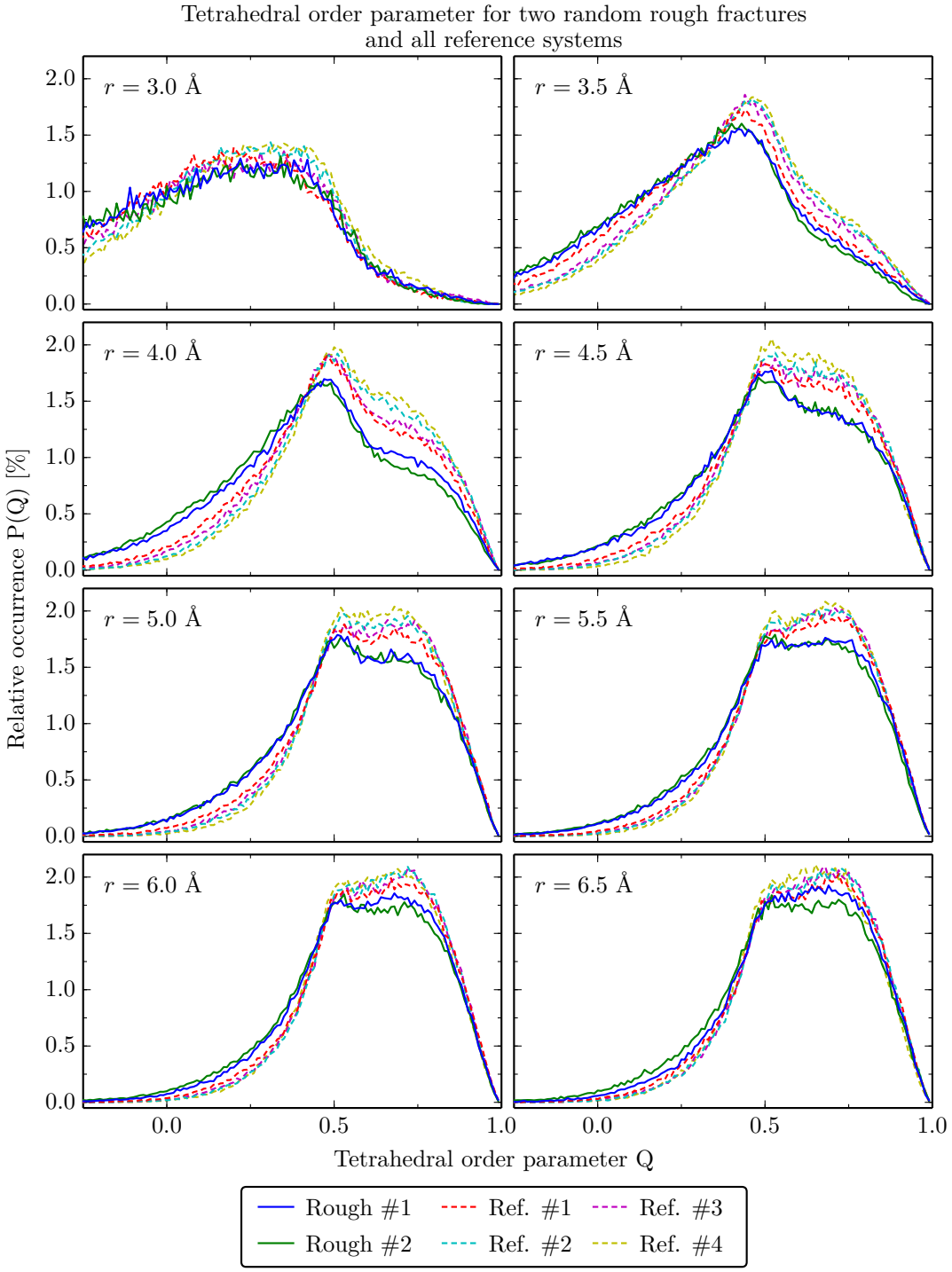


Figure 9.8: Plots of $P(Q)$ for the two random rough fractures ("rough" #1 and #2, solid lines), and all four reference systems (dashed lines).

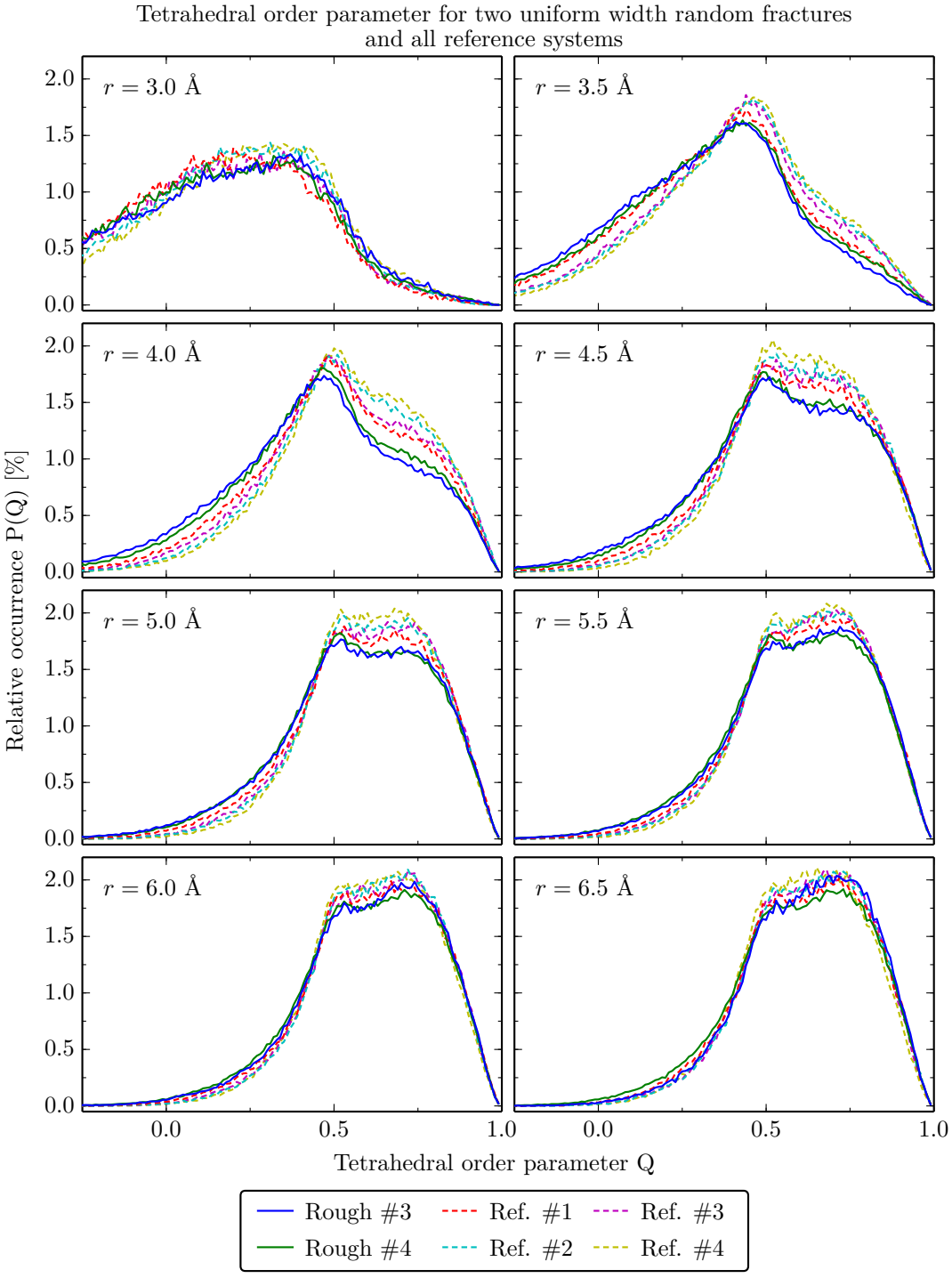


Figure 9.9: Plots of $P(Q)$ for the two random rough fractures of uniform width (solid lines), with fractures respectively 14.4 Å ("rough #3", blue solid line) and 28.8 Å ("rough #4", green solid line) wide, and all four reference systems (the dashed lines).

see that the two peaks near $Q = 0.5$ and $Q = 0.75$ seems to be lower for the rough fractures than the flat pores, at least up to 5.5 Å. As for the random rough fractures, there are again no clear differences between the two rough fracture systems with uniform width, other than a slightly higher peak for rough fracture system #3 at 6.5 Å.

9.4 Distance to nearest atom

We have made 3d maps of the system labelled "rough fracture #2", using the method from section 7.6, which finds the distance to the nearest atom on a grid of points. The results can be seen in figs. 9.10a and 9.10b, where we show slices of the maps in the yz- and xy-plane. For the first figure we used a max distance of 5 Å, while for the second one we used a max distance of 20 Å. The maps were made from the molecular system after cutting out the atoms to make the pore and passivating the dangling ends, but before filling the pore with water. A similar map can me made by not including the water molecules in the calculations.

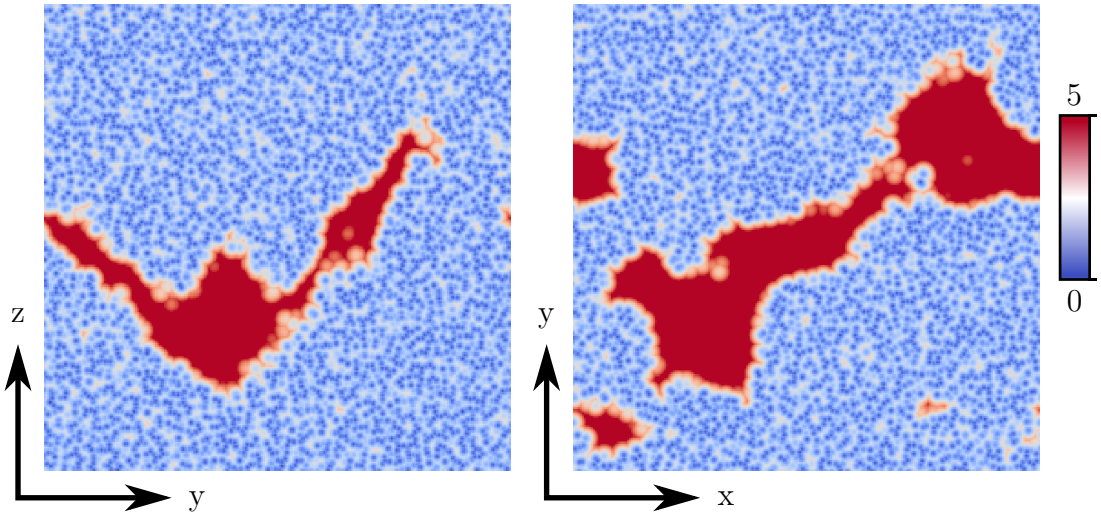
In fig. 9.10a we can clearly see the positions of the atoms in the silica matrix as the dark blue dots, while the pores light up as red areas.

In fig. 9.10b we still see the positions of the atoms, but they are less visible now since we have a bigger range for the colormap. We can still see the pores easily, colored white, and we now also see some characteristics of the pore itself, where it has a darker red color.

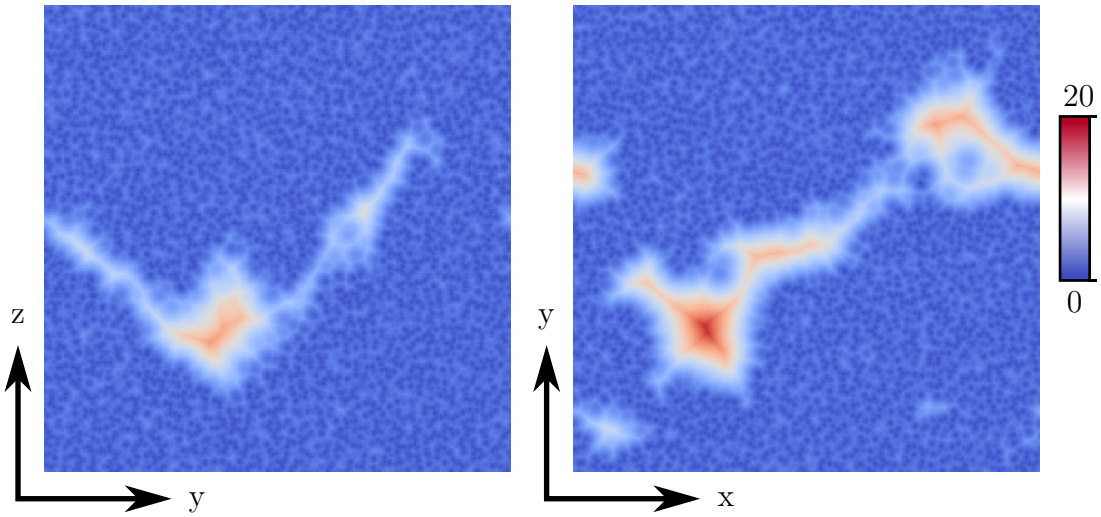
9.5 Manhattan distance to nearest atom

We have made 3d maps of the system labelled "rough fracture #2", of the Manhattan distance to the nearest atom to each point on a grid, using the method from section 7.7. See figs. 9.11a and 9.11b for the results, where we show slices of the maps in the yz- and xy-plane.

We see that the 3d maps made using this method shows a lot of the same details as the maps in figs. 9.10a and 9.10b, but we see that using the Manhattan distance can make the results harder to interpret, since the Manhattan distance between two points is generally shorter than the Euclidean distance, and we are not used to interpreting the Manhattan distance. In molecular dynamics simulations it is also unusual to use the Manhattan distance.

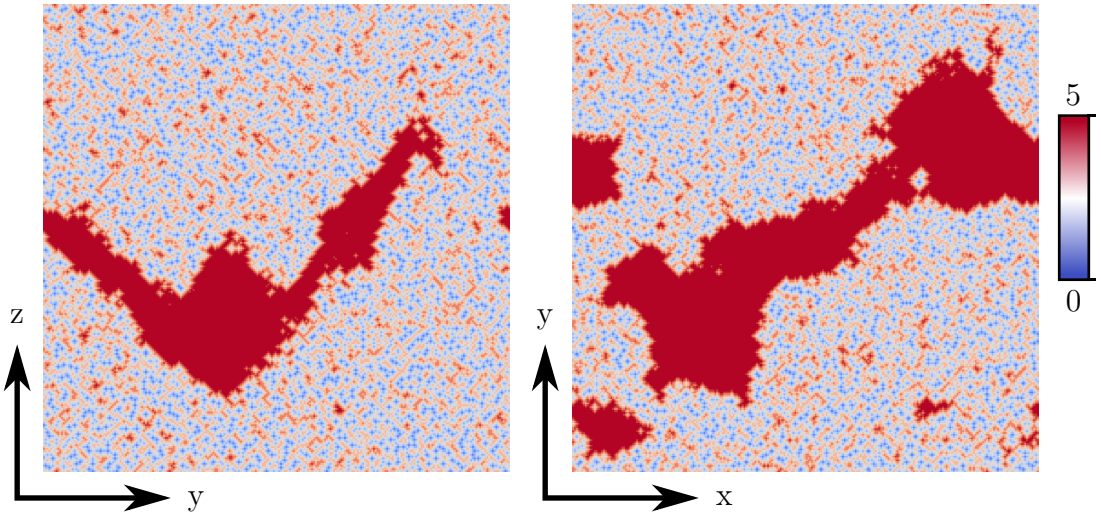


(a) Max distance $r_{\max} = 5.0$ Å.

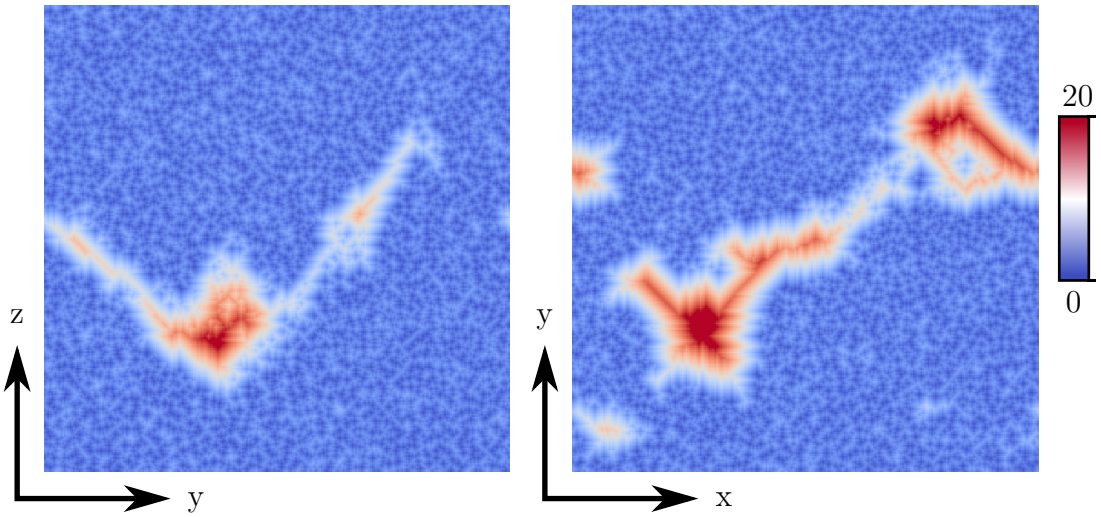


(b) Max distance $r_{\max} = 20$ Å.

Figure 9.10: Slices of 3d maps of the distance to the nearest atom in the system labelled "rough fracture #2", generated using method from section 7.6, using a colormap that goes from **(a)** 0 to 5 Å, and **(b)** 0 to 20 Å.



(a) Max distance $r_{\max} = 5.0$ Å.



(b) Max distance $r_{\max} = 20$ Å.

Figure 9.11: Slices of 3d maps of the Manhattan distance to the nearest atom, in the system labelled "rough fracture #2", made using the method from section 7.7. We have used colormaps from 0 to $r_{\max}$, with **(a)** $r_{\max} = 8$ Å, and **(b)** $r_{\max} = 30$ Å.

Chapter 10

Discussion, conclusions and future

10.1 Discussion

After doing a thorough study of the measurements done in the simulations, we will now try to draw some conclusions from the different results we have found. We will first discuss the individual results from each physical quantity we have measured, before we try to put these results into perspective.

The results of the measurements of the water density in the different simulated systems shows that some of our systems has much higher density than the other systems, especially refence system #2-4. When filling the pores in these systems with water we used a higher input density, so this was according to our intention, and shows that the method we use for filling pores with water works as intended for those systems. But we also used much higher input densities in rough fracture system #3 and #4 than #1 and #2, and we did not measure higher water density in those systems. This can be explained by looking at the geometry of the fractures in system #3 and #4, which are very narrow fractures, respectively 14.4 and 28.8 Å wide. The method we use for filling the pores with water first divides the system into cuboid voxels (with the size of the voxels calculated from the wanted density), marks all voxels with an atom in them as occupied, and then puts one water atom in each unoccupied voxel. This voxelation approach has trouble when we use it on rough surfaces, as it will mark a lot of the voxels near the surface as occupied, even though we in reality could fit several water molecules near the surface. In a narrow pore, with a lot of surface area compared to the pore volume, a lot of the voxels near the surface will be marked as occupied, and the result is that we get a lower water density then we wanted.

When we look at the water density as function of distance to the silica matrix we see the same behaviour in all systems, independently of the structure of the pores and fractures. The density peaks at around 7-8 Å from the silica matrix, and drop by almost 10% at 5 Å. This is a clear indication of interactions between the silica matrix and the water molecules.

We also saw a trend in the peak in density, which appeared around 7-8 Å from the silica matrix, but which seemed to move closer to the silica matrix as we increased the bulk density. It seems like the increased water pressure makes the water molecules retain bulk-like behaviour closer to the silica surface.

In the measurements of the diffusion we saw clear changes in the transport properties as we got close to the silica matrix. We saw the same behaviour in all simulations, independent of the pore geometry. The diffusion constant was stable up to around 7 Å from the silica matrix, where it made a dip near 5.5 Å, before reaching a local peak at 5 Å, and then going linearly to zero from 5 to 3 Å. We again see clear results of interactions between the water molecules and the silica matrix. We know that silica is hydrophilic, so when the diffusion goes to zero the cause can be that the water molecules are attracted to the silica surface, which slows down the self diffusion. We will also have an effect of water molecules colliding with the silica surface, which further slows down diffusion.

When comparing reference systems #1 and #2 which have the exact same pore geometry and dimensions, we see a clearly reduced diffusion in one of the systems. As the only difference between those two systems is the density, we see an indication an increase in density lowers self diffusion. When we increase density we expect collisions between water molecules to happen more often, reducing the mean free path of the molecules, and slowing down diffusion.

Although the diffusion all systems had very similar behaviour, rough fracture system #3, which is a 14.4 Å narrow rough fracture, displayed a somewhat reduced diffusion constant at all distances from the matrix, compared to the other rough fracture systems. We should compare this to to rough fracture system #4, which has a similar fracture, but twice as wide (28.8 Å). This system shows no reduction in the diffusion, but almost perfectly matches the diffusion in a 86 Å wide flat pore (reference system #1). We also compare it to reference system #3, which has a 14.4 Å flat pore. This system has very similar diffusion to rough fracture system #3, although this system also has a somewhat higher water density, which may cause reduced diffusion. We conclude with that the reduction in diffusion in rough fracture system #3 is caused by the width of the fracture itself, since the water density in this system is similar to the density in the other rough fracture systems.

When measuring the tetrahedral order parameter we saw similar behaviour in all systems, independent of pore geometry and structure. We observed clear changes

in the structure of the water molecules as we go close to the silica matrix. At 5.5 Å and further away from the matrix the water had close to bulk properties in all cases, but as we moved closer than this one of the observed peaks in the distribution was reduced, by almost a factor 2 by the time we get to 3.5 Å. This is a clear indication that the silica matrix is affecting the internal structure in the water, and the arrangement of the water molecules. This can again be caused by the hydrophilic silica, which can make the water molecules arrange themselves in a structure similar to the nearby silica structure. Since water has strong hydrogen bonds between water molecules, this effect and arrangement can be retained several layers of water molecules from the silica matrix, beyond the reach of the force from the silica itself.

Although the tetrahedral order parameter was very similar for all systems independent of pore geometry and other characteristics, we see that the reference systems are more similar to each other than to the rough fracture systems, and that the rough fracture systems are more similar to each other than to the reference systems. The only real deviation we see is in rough fracture system #3, which is a 14.4 Å narrow fracture. For this system we see that one of the peaks in the distribution of tetrahedral order paramters seems to rise a bit faster when we increase the distance to the silica matrix, compared to the other rough fracture systems.

10.2 Conclusions

To investigate the structure of water trapped in pores in nanoporous silica we have measured the density and tetrahedral order parameter of water, as function of the distance to the silica matrix. We found that the structure and density of water changed drastically as we got within 6-8 Å of the silica matrix.

When measuring the water density we saw that it seemed to decrease by around 10% of the bulk density when we got to around 5 Å from the silica matrix, before increasing to at higher than the the bulk density at 3 Å from the matrix. The minimum point around 5 Å seemed roughly constant for all pore geometries and bulk densities, but the falloff for the density, around 7 Å seemed to move closer to the silica matrix as we increased the bulk density.

When we measured the tetrahedral order parameter for water molecules, a number between 0 and 1 that tells us something about how “tetrahedral” a set of four points is, we saw that the water molecules had the expected structure in bulk conditions (more than 10 Å from a silica matrix), but that as we got to 5.5 Å and closer to the silica matrix this parameter was disturbed by the silica matrix, and we saw what appeared to be changes in the intermolecular structure.

This change in the structure and density of the water molecules can be caused by the hydrophilic nature of silica, and the interactions between the silica matrix and the water molecules. What may happen is that the water molecules close to the silica matrix are attracted to the matrix, and they arrange themselves in a structure similar to the silica structure. Since the water molecules has strong hydrogen bonds between them, this change in structure will be retained several layers of water molecules into the water, beyond the reach of the water-silica interaction.

To study the transport properties of water in nanoporous silica we measured the self-diffusion constant of the water, both in bulk-like water far from the silica surface, and as function of the distance to the silica surface, closer than 10 Å from the silica matrix. We found that the diffusion was approximately equal to the bulk at up to around 7 Å from the silica matrix, where it started decreasing. The diffusion constant decreased approximately linearly from 5 to 3 Å from the silica matrix, at which point all diffusion stopped. This behaviour can again be caused by the surface interaction between water and silica, where the silica attract the water molecules, and hinders the movement and diffusion. This attraction causes the water molecules to arrange in a certain way near the silica surface, which again affects the next layers of water molecules via the strong hydrogen bonds between water molecules, and we see a reduction in diffusion several up 7 Å from the silica matrix.

When measuring diffusion we also noticed that the self diffusion decreased with increasing water density, which can be caused by the reduced mean free path in water with increased density.

Perhaps the most interesting thing we saw during our simlulations was that the geometries of the fractures and pores had little to no effect on the water, as water in both completely flat pores and in very rough and random fractures seemed to exhibit almost the same characteristics, both in transport properties and structure. The system that deviated the most from this was the system with a narrow 14.4 Å rough fracture, which had about the same water density as the other rough fracture systems, but showed a somewhat reduced diffusion at all distances from the matrix, compared to the other rough fractures. This system also showed some differences in the tetrahedral order parameter.

10.3 Future

When doing measurements on the structure and transport properties of water we were surprised to see that the simulations we did showed little differences, even though the structure and characteristics of the different systems we simulated

were very different. We saw some interesting differences when simulating systems with pores as narrow as 14.4 Å, and a more extensive and thorough study of pores of this size, and smaller, could turn out to be fruitful.

One big problem if one wants to study narrow pores is, as we discovered, controlling the density of water in the pores. The method we developed and used for filling pores and fractures with water works good in large fractures and far from the silica surface, but in narrow fractures with a lot of silica surface the voxelation technique breaks down, and we are unable to insert the number of water molecules needed to get the density we want. To do further studies of water in nanoporous silica one should thus try to improve this method for filling pores with water, or develop a completely new method for filling narrow pores with water. A grand canonical Monte Carlo based approach might work well here, or perhaps some way of starting to fill the pore with water at the silica surface, so the surface is properly saturated.

One of the hurdles in studying any surface, or a fractured or porous system, is defining *where* the surface actually *is*. We circumvented this problem by defining the distance to the surface as the distance to the nearest silicon atom, since we only needed the distance to the surface. This works well in a lot of cases, but partly breaks down when we go to distances similar to or smaller than the average distances between the atoms the surface consist of (closer than about 3 Å in our case). A different solution is to define the surface as a set of mathematical planes, and the most practical thing to do here is probably to define the surface as sets of *triangles*. Triangulation of surfaces and volumes is a method studied a lot both by mathematicians and by computer graphics developers, so there are a lot of efficient methods for doing calculations on these kinds of data. Although a method based on triangles seem good at first glance, one must ask if it is even possible to define the surface of something as a plane at an atomic scale. Because at those scales a surface actually consist of atoms and molecules, and not simple planes.

An improvement that could improve on our results would be to develop a method that generates fractures with actual uniform width across the whole system, since the "uniform width" fractures we simulated only really had uniform distance between the two surfaces in the z-direction, and thus the actual width of the pore varied throughout the system, although it is limited by and related to the distance between the two surfaces.

Lastly we would have liked to measure the diffusion normal to and parallell to the silica surfaces in our systems. As this would require us to locate and define the surface, which we have seen is a complicated problem, we were unable to do this.

Part IV

Appendices

Appendix A

Verlet integrators

The Verlet algorithm[45] is a simple method for numerically integrating second order differential equations of the form

$$\frac{\mathrm{d}^2\boldsymbol{r}(t)}{\mathrm{d}t^2} = \boldsymbol{F}\big[\boldsymbol{r}(t), t\big] = \boldsymbol{F}(t).$$

The algorithm has several equivalent forms, and the form originally used by Verlet is

$$\boldsymbol{r}(t+\Delta t) \approx 2\boldsymbol{r}(t) - \boldsymbol{r}(t-\Delta t) + \boldsymbol{a}(t)\Delta t^2,$$

where Δt is the timestep, and $\boldsymbol{a}(t)$ is the velocity at time t. An equivalent formulation, usually called the velocity Verlet algorithm, has the form

$$\begin{aligned}\boldsymbol{r}(t+\Delta t) &= \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \frac{\boldsymbol{F}(t)}{2m}\Delta t^2 \\ \boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \frac{\boldsymbol{F}(t+\Delta t) + \boldsymbol{F}(t)}{2m}\Delta t.\end{aligned}$$

The velocity Verlet algorithm is the most used form of the algorithm, and it has a accumulated error of $\mathcal{O}(\Delta t^2)$, as we show in appendix A.2.3.

We will now first derive the regular Verlet and the velocity Verlet algorithms using Taylor expansions, and then using the Liouville formulation of classical mechanics.

A.1 Deriving the Verlet algorithm using Taylor expansions

To derive the algorithm we first let

$$\frac{\mathrm{d}\boldsymbol{r}(t)}{\mathrm{d}t} = \boldsymbol{v}(t),$$

and

$$\frac{\mathrm{d}\boldsymbol{v}(t)}{\mathrm{d}t} = \boldsymbol{a}(t) = \frac{\boldsymbol{F}(t)}{m}.$$

We then do a Taylor expansion of $\boldsymbol{r}(t \pm \Delta t)$ around time t

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2} + \frac{\mathrm{d}^3\boldsymbol{r}(0)}{\mathrm{d}t^3}\frac{\Delta t^3}{6} + \mathcal{O}(\Delta t^4), \tag{A.1}$$

$$\boldsymbol{r}(t-\Delta t) = \boldsymbol{r}(t) - \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2} - \frac{\mathrm{d}^3\boldsymbol{r}(0)}{\mathrm{d}t^3}\frac{\Delta t^3}{6} + \mathcal{O}(\Delta t^4). \tag{A.2}$$

By summing these two equations we get

$$\boldsymbol{r}(t+\Delta t) + \boldsymbol{r}(t-\Delta t) = 2\boldsymbol{r}(t) + \boldsymbol{a}(t)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

which by rearranging can be written as

$$\boldsymbol{r}(t+\Delta t) \approx 2\boldsymbol{r}(t) - \boldsymbol{r}(t-\Delta t) + \boldsymbol{a}(t)\Delta t^2.$$

This is the equation used to update the positions in the regular Verlet algorithm. We see that the estimate of the new position contains an truncation error for one timestep Δt of the order $\mathcal{O}(\Delta t^4)$.

The Verlet algorithm does not use the velocity to compute the new position, but we can find an estimate of the velocity by taking the difference between eqs. (A.1) and (A.2)

$$\boldsymbol{r}(t+\Delta t) - \boldsymbol{r}(t-\Delta t) = 2\boldsymbol{v}(t)\Delta t + \mathcal{O}(\Delta t^3),$$

which by rearranging can be written as

$$\boldsymbol{v}(t) = \frac{\boldsymbol{r}(t+\Delta t) - \boldsymbol{r}(t-\Delta t)}{2\Delta t} + \mathcal{O}(\Delta t^2).$$

We see that this estimate of the velocity has a truncation error of the order $\mathcal{O}(\Delta t^2)$, compared to the error in the position $\mathcal{O}(\Delta t^4)$.

A.1.1 Velocity Verlet

A modification of the Verlet algorithm usually called the velocity Verlet algorithm can be derived in a similar way. We have the same Taylor expansion of $\boldsymbol{r}(t+\Delta t)$ around t as before

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3), \tag{A.3}$$

and now we also expand $\boldsymbol{v}(t+\Delta t)$ around t

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \boldsymbol{a}(t)\Delta t + \frac{\mathrm{d}^2\boldsymbol{v}(t)}{\mathrm{d}t^2}\,\frac{\Delta t^2}{2} + \mathcal{O}(\Delta t^3). \tag{A.4}$$

We now need an expression for $\frac{\mathrm{d}^2\boldsymbol{v}(t)}{\mathrm{d}t^2}$, which can be found by a Taylor expansion of $\frac{\mathrm{d}\boldsymbol{v}(t+\Delta t)}{\mathrm{d}t}$

$$\frac{\mathrm{d}\boldsymbol{v}(t+\Delta t)}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{v}(t)}{\mathrm{d}t} + \frac{\mathrm{d}^2\boldsymbol{v}(t)}{\mathrm{d}t^2}\,\Delta t + \mathcal{O}(\Delta t^2),$$

which by rearranging and multiplying with $\frac{\Delta t}{2}$ gives

$$\begin{aligned}\frac{\mathrm{d}^2\boldsymbol{v}}{\mathrm{d}t^2}\,\frac{\Delta t^2}{2} &= \left(\frac{\mathrm{d}\boldsymbol{v}(t+\Delta t)}{\mathrm{d}t} - \frac{\mathrm{d}\boldsymbol{v}(t)}{\mathrm{d}t}\right)\frac{\Delta t}{2} + \mathcal{O}(\Delta t^3)\\ &= \big[\boldsymbol{a}(t+\Delta t) - \boldsymbol{a}(t)\big]\frac{\Delta t}{2} + \mathcal{O}(\Delta t^3).\end{aligned}$$

Inserting this into eq. (A.4) we get

$$\begin{aligned}\boldsymbol{v}(t+\Delta t) &= \boldsymbol{v}(t) + \boldsymbol{a}(t)\Delta t + \big[\boldsymbol{a}(t+\Delta t) - \boldsymbol{a}(t)\big]\frac{\Delta t}{2} + \mathcal{O}(\Delta t^3)\\ &= \boldsymbol{v}(t) + \big[\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t)\big]\frac{\Delta t}{2} + \mathcal{O}(\Delta t^3).\end{aligned} \tag{A.5}$$

So the total velocity Verlet algorithm with truncation of the higher-order terms is

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \boldsymbol{a}(t)\frac{\Delta t^2}{2} \tag{A.6}$$

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \big[\boldsymbol{a}(t) + \boldsymbol{a}(t+\Delta t)\big]\frac{\Delta t}{2}, \tag{A.7}$$

with the truncation error for one timestep Δt being of the order $\mathcal{O}(\Delta t^3)$ for both the position and the velocity.

A.2 Deriving velocity Verlet using Liouville operator

We will now derive the velocity Verlet algorithm in a more rigorous way, using the Liouville formulation of classical mechanics. This approach will give us better insight into why the algorithm is so powerful, and a good estimate of the global (or accumulated) error of this algorithm. The derivation closely follows section 4.3.3 in [13].

A.2.1 Liouville operator

We have a system consisting of N particles, with positions $\boldsymbol{r}$ and momenta $\boldsymbol{p}$. We define a function of these variables $f(\boldsymbol{r}(t), \boldsymbol{p}(t)) = f(t)$, that has the time derivative (denoted by a dot)

$$\dot{f}(t) = \dot{\boldsymbol{r}} \frac{\partial f(t)}{\partial \boldsymbol{r}} + \dot{\boldsymbol{p}} \frac{\partial f(t)}{\partial \boldsymbol{p}} = i\hat{\boldsymbol{L}} f(t). \tag{A.8}$$

where we have defined the *Liouville operator*, $i\hat{\boldsymbol{L}}$, as

$$i\hat{\boldsymbol{L}} = \dot{\boldsymbol{r}} \frac{\partial}{\partial \boldsymbol{r}} + \dot{\boldsymbol{p}} \frac{\partial}{\partial \boldsymbol{p}} = i\hat{\boldsymbol{L}}_r + i\hat{\boldsymbol{L}}_p$$

where $i\hat{\boldsymbol{L}}_r$ and $i\hat{\boldsymbol{L}}_p$ denotes the left and right part if this operator, respectively. We can formally integrate eq. (A.8) to obtain

$$f(t) = e^{i\hat{\boldsymbol{L}}t} f(0), \tag{A.9}$$

which allows us to define the time evolution operator $\hat{\mathcal{U}} = \exp(i\hat{\boldsymbol{L}}t)$. We see that this integration doesn't get us any closer to finding $f(t)$, since evaluating the right-hand side is equivalent to the exact integration of the classical equations of motion. To get around this we define the time evolution operator for positions $\hat{\mathcal{U}}_r(t) = \exp(i\hat{\boldsymbol{L}}_r t)$, and try applying just this operator. If we do a Taylor expansion of the exponential we get

$$\begin{aligned}
\hat{\mathcal{U}}_r(t) f(0) &= f(0) + i\hat{\boldsymbol{L}}_r t f(0) + \frac{(i\hat{\boldsymbol{L}}_r t)^2}{2!} f(0) + \dots \\
&= \exp\left(\dot{\boldsymbol{r}}(0) t \frac{\partial}{\partial \boldsymbol{r}} \right) f(0) \\
&= \sum_{n=0}^{\infty} \frac{\left(\dot{\boldsymbol{r}}(0) t\right)^n}{n!} \frac{\partial^n}{\partial \boldsymbol{r}^n} f(0) \\
&= f\left\{ \left[\boldsymbol{r}(0) + \dot{\boldsymbol{r}}(0) t \right], \boldsymbol{p}(0) \right\},
\end{aligned}$$

where $\boldsymbol{r}(0)$ and $\boldsymbol{p}(0)$ are the positions and momenta at $t = 0$. We see that this has the effect of moving the positions $\boldsymbol{r}$ a step t forward in time according to their derivative. It's easy to see that the equivalent momentum time evolution operator $\hat{\mathcal{U}}_p(t) = \exp(i\hat{\boldsymbol{L}}_p t)$ has a similar effect on the momenta.

A.2.2 Velocity Verlet

In a molecular dynamics simulation we would like to be able to apply these operators independently, since

$$\hat{\mathcal{U}} = e^{i\hat{\boldsymbol{L}}} = e^{i\hat{\boldsymbol{L}}_r + i\hat{\boldsymbol{L}}_p},$$

but unfortunately, for two noncommuting operators $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$ we have

$$e^{\hat{\boldsymbol{A}}+\hat{\boldsymbol{B}}} \neq e^{\hat{\boldsymbol{A}}} e^{\hat{\boldsymbol{B}}}.$$

To solve this we can use the following *Trotter identity*

$$e^{\hat{\boldsymbol{A}}+\hat{\boldsymbol{B}}} = \lim_{P\to\infty} \left(e^{\hat{\boldsymbol{A}}/2P} e^{\hat{\boldsymbol{B}}/P} e^{\hat{\boldsymbol{A}}/2P} \right)^P.$$

Applying the operators an infinite number of times ($P \to \infty$) is unpractical, but fortunately the expression can be truncated for large but finite P as

$$e^{\hat{\boldsymbol{A}}+\hat{\boldsymbol{B}}} = \left(e^{\hat{\boldsymbol{A}}/2P} e^{\hat{\boldsymbol{B}}/P} e^{\hat{\boldsymbol{A}}/2P} \right)^P e^{\mathcal{O}(1/P^2)}. \tag{A.10}$$

To derive the velocity Verlet scheme using this truncation we first identify the operators $\hat{\boldsymbol{A}}$ and $\hat{\boldsymbol{B}}$ as

$$\begin{aligned} \frac{\hat{\boldsymbol{A}}}{P} &\equiv \frac{i\hat{\boldsymbol{L}}_p t}{P} \equiv \Delta t \dot{\boldsymbol{p}}(0) \frac{\partial}{\partial \boldsymbol{p}} \\ \frac{\hat{\boldsymbol{B}}}{P} &\equiv \frac{i\hat{\boldsymbol{L}}_r t}{P} \equiv \Delta t \dot{\boldsymbol{r}}(0) \frac{\partial}{\partial \boldsymbol{r}} \end{aligned}$$

where $\Delta t = t/P$. We can now identify the *truncated* time evolution operator $\hat{\mathcal{U}}^*(t)$ as follows

$$\begin{aligned} \hat{\mathcal{U}}(t) &= \left(e^{i\hat{\boldsymbol{L}}_p \Delta t/2} e^{i\hat{\boldsymbol{L}}_r \Delta t} e^{i\hat{\boldsymbol{L}}_p \Delta t/2} \right)^P e^{\mathcal{O}(1/P^2)} \\ &\approx \left(e^{i\hat{\boldsymbol{L}}_p \Delta t/2} e^{i\hat{\boldsymbol{L}}_r \Delta t} e^{i\hat{\boldsymbol{L}}_p \Delta t/2} \right) = \hat{\mathcal{U}}^*(t), \end{aligned} \tag{A.11}$$

and the the truncated operator for moving *one* timestep forward as

$$\hat{\mathcal{U}}^*(\Delta t) = e^{i\hat{\boldsymbol{L}}_p \Delta t/2} e^{i\hat{\boldsymbol{L}}_r \Delta t} e^{i\hat{\boldsymbol{L}}_p \Delta t/2}. \tag{A.12}$$

To see the effect of the operator $\hat{\mathcal{U}}^*(\Delta t)$ on the coordinates and momenta of the particles we first apply $\exp(i\hat{\boldsymbol{L}}_p\Delta t/2)$ to $f(0)$, and get

$$e^{i\hat{\boldsymbol{L}}_p\Delta t/2}f(0) = f\left\{\boldsymbol{r}(0), \left[\boldsymbol{p}(0) + \frac{\Delta t}{2}\dot{\boldsymbol{p}}(0)\right]\right\}.$$

We then apply $\exp(i\hat{\boldsymbol{L}}_r\Delta t)$ and get

$$e^{i\hat{\boldsymbol{L}}_r\Delta t}f\left\{\boldsymbol{r}(0), \left[\boldsymbol{p}(0) + \frac{\Delta t}{2}\dot{\boldsymbol{p}}(0)\right]\right\} = f\left\{\left[\boldsymbol{r}(0) + \Delta t\dot{\boldsymbol{r}}\left(\frac{\Delta t}{2}\right)\right], \left[\boldsymbol{p}(0) + \frac{\Delta t}{2}\dot{\boldsymbol{p}}(0)\right]\right\},$$

and finally we apply $\exp(i\hat{\boldsymbol{L}}_p\Delta t/2)$ once more, and get

$$f\left\{\left[\boldsymbol{r}(0) + \Delta t\dot{\boldsymbol{r}}\left(\frac{\Delta t}{2}\right)\right], \left[\boldsymbol{p}(0) + \frac{\Delta t}{2}\dot{\boldsymbol{p}}(0) + \frac{\Delta t}{2}\dot{\boldsymbol{p}}(\Delta t)\right]\right\}.$$

If we look at the total effect of applying the operator we see that

$$\boldsymbol{r}(0) \to \boldsymbol{r}(0) + \Delta t\dot{\boldsymbol{r}}\left(\frac{\Delta t}{2}\right) \tag{A.13}$$

$$\boldsymbol{p}(0) \to \boldsymbol{p}(0) + \left[\dot{\boldsymbol{p}}(0) + \dot{\boldsymbol{p}}(\Delta t)\right]\frac{\Delta t}{2}. \tag{A.14}$$

Using the relations $\boldsymbol{p} = m\boldsymbol{v}$, $\dot{\boldsymbol{p}} = \boldsymbol{F}$, and, if we assume that the forces only depend on the positions, $\boldsymbol{F}(\boldsymbol{r}(t)) = \boldsymbol{F}(t)$, the relation

$$\dot{\boldsymbol{r}}(\Delta t/2) = \boldsymbol{r}(0) + \frac{\boldsymbol{F}(0)}{m}\frac{\Delta t}{2},$$

we can rewrite eqs. (A.13) and (A.14) to

$$\boldsymbol{v}(\Delta t) = \boldsymbol{v}(0) + \left[\frac{\boldsymbol{F}(0)}{m} + \frac{\boldsymbol{F}(\Delta t)}{m}\right]\frac{\Delta t}{2}$$

$$\boldsymbol{r}(\Delta t) = \boldsymbol{r}(0) + \boldsymbol{v}(0)\Delta t + \frac{\boldsymbol{F}(0)}{m}\frac{\Delta t^2}{2},$$

which is exactly the velocity Verlet algorithm, as we saw in eqs. (1.3) and (1.4).

A.2.3 Error in velocity Verlet

When we approximate the exact time evolution operator for one timestep $\hat{\mathcal{U}}(\Delta t)$ with $\hat{\mathcal{U}}^*(\Delta t)$, going from eq. (A.10) to eqs. (A.11) and (A.12), we do a truncation

$$\hat{\mathcal{U}}(\Delta t) = \hat{\mathcal{U}}^*(\Delta t)e^{\mathcal{O}(1/P^2)} \approx \hat{\mathcal{U}}^*(\Delta t).$$

To investigate the error introduced by this truncation we express it as an *error operator* $\hat{\epsilon}$

$$e^{i\hat{\boldsymbol{L}}_p \Delta t/2} e^{i\hat{\boldsymbol{L}}_r \Delta t} e^{i\hat{\boldsymbol{L}}_p \Delta t/2} e^{\mathcal{O}(1/P^2)} = e^{i\hat{\boldsymbol{L}}\Delta t + \hat{\epsilon}},$$

which, using Campbell-Baker-Hausdorff expansion, can be expressed in terms of the commutators of $\boldsymbol{L}_p$ and $\boldsymbol{L}_r$ as

$$\hat{\epsilon} = \sum_{n=1}^{\infty} (\Delta t)^{2n+1} c_{2n+1}, \tag{A.15}$$

where c_m denotes a combination of mth-order commutators.

From this it can be shown that, if the expansion in eq. (A.15) converges, Verlet style integrators will rigorously conserve a *pseudo*-Hamiltonian, and that the difference between this pseudo-Hamiltonian and the actual Hamiltonian is of the order $(\Delta t)^{2n}$, where n depends on the order of the algorithm[13, section 4.3.3].

Appendix B

Nosé-Hoover thermostats

We will here derive the Nosé-Hoover thermostat. We closely follow the derivation in section 6.1.2 [13]. See also [31, 16] for more information.

To show how the thermostat works we need to use the Lagrangian and Hamiltonian formulation of classical mechanics. The Lagrangian $\mathcal{L}$ of a classical N-body system is defined as the kinetic energy minus the potential energy U

$$\mathcal{L} = K - U$$

and what is called the *generalized* momentum $\boldsymbol{p}_i$ of a *generalized* coordinate $\boldsymbol{q}_i$ defined as

$$\boldsymbol{p}_i = \frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{q}}_i}, \tag{B.1}$$

where we denote the time derivative by a dot. These generalized coordinates and momenta are not bound to any one coordinate system, and may be any quantitative attribute of the system.

What the Nosé-Hoover thermostat does is to introduce an additional coordinate s to the Lagrangian, creating a virtual, extended system, with the following Lagrangian[31]:

$$\mathcal{L}_{\text{Nosé}} = \sum_{i=1}^{N} \frac{m_i}{2} s^2 \dot{\boldsymbol{r}}_i^2 - U(\boldsymbol{r}) + \frac{Q}{2}\dot{s}^2 - 3Nk_BT \ln s, \tag{B.2}$$

where Q is an effective "mass" associated with s, and $\boldsymbol{r}_i$ is the *generalized* coordinate from earlier, interpreted as a virtual position of an atom using cartesian coordinates. The momenta of this virtual system follow from eqs. (B.1) and (B.2)

as

$$\begin{aligned} \boldsymbol{p}_i &= \frac{\partial \mathcal{L}}{\partial \dot{\boldsymbol{r}}_i} = m_i s^2 \dot{\boldsymbol{r}}_i \\ p_s &= \frac{\partial \mathcal{L}}{\partial \dot{s}} = Q\dot{s}. \end{aligned}$$

This gives the following Hamiltonian for the extended system

$$\mathcal{H}_{\text{Nosé}} = \sum_{i=1}^{N} \frac{\boldsymbol{p}_i^2}{2m_i s^2} + U(\boldsymbol{r}) + \frac{p_s^2}{2Q} + 3Nk_BT \ln s, \tag{B.3}$$

It can be shown that we can relate the generalized coordinates to real variables (real variables indicated by a prime) as follows

$$\begin{aligned} \boldsymbol{r}' &= \boldsymbol{r} \\ \boldsymbol{p}' &= \boldsymbol{p}/s \\ s' &= s \\ \Delta t' &= \Delta t/s. \end{aligned} \tag{B.4}$$

From eq. (B.4) we see that s can be interpreted as a scaling factor of the time step.

From the Hamiltonian eq. (B.3) we can derive the equations of motion for the virtual variables $\boldsymbol{r}$, $\boldsymbol{p}$, and t, and the real variables $\boldsymbol{r}'$, $\boldsymbol{p}'$, and t' [16]

$$\begin{aligned} \frac{\mathrm{d}\boldsymbol{r}_i'}{\mathrm{d}t'} &= s\frac{\mathrm{d}\boldsymbol{r}_i}{\mathrm{d}t} = \frac{\boldsymbol{p}_i}{m_i s} = \frac{\boldsymbol{p}_i'}{m_i} \\ \frac{\mathrm{d}\boldsymbol{p}_i'}{\mathrm{d}t'} &= s\frac{\mathrm{d}(p_i/s)}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{p}_i}{\mathrm{d}t} - \frac{1}{s}\boldsymbol{p}_i\frac{\mathrm{d}s}{\mathrm{d}t} = -\frac{\partial U(\boldsymbol{r}')}{\partial \boldsymbol{r}_i'} - \frac{s'p_s'}{Q}\boldsymbol{p}_i' \\ \frac{1}{s}\frac{\mathrm{d}s'}{\mathrm{d}t'} &= \frac{s}{s}\frac{\mathrm{d}s}{\mathrm{d}t} = \frac{s'p_s'}{Q} \\ \frac{\mathrm{d}(s'p_s'/Q)}{\mathrm{d}t'} &= \frac{s}{Q}\frac{\mathrm{d}p_s}{\mathrm{d}t} = \left(\sum_{i=1}^{N}\frac{p_i'^2}{m_i} - 3Nk_BT\right)/Q. \end{aligned}$$

These equations can further be simplified if we introduce a thermodynamic friction coefficient $\xi = s'p_s'/Q$ and drop the primes. We then get the following

equations of motion

$$\xi = \frac{sp_s}{Q} \tag{B.5}$$

$$\dot{\boldsymbol{r}}_i = \frac{\boldsymbol{p}_i}{m} \tag{B.6}$$

$$\dot{\boldsymbol{p}}_i = -\frac{\partial U(\boldsymbol{r})}{\partial r_i} - \xi \boldsymbol{p}_i \tag{B.7}$$

$$\dot{\xi} = \left(\sum_{i=1}^{N} \frac{p_i^2}{m_i} - 3Nk_BT \right) /Q \tag{B.8}$$

List of Figures

List of Tables

List of listings

Bibliography

[1] P. Abry and F. Sellan. "The wavelet-based synthesis for fractional Brownian motion proposed by F. Sellan and Y. Meyer: Remarks and fast implementation". In: *Applied and computational harmonic analysis* 383 (1996), pp. 377–383.

[2] E. Alessio et al. "Second-order moving average and scaling of stochastic time series". In: *The European Physical Journal B - Condensed Matter* 27.2 (2002), pp. 197–200.

[3] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Clarendon Press, 1989.

[4] H. J. C. Berendsen et al. "Molecular dynamics with coupling to an external bath". In: *The Journal of Chemical Physics* 81.8 (1984), p. 3684.

[5] P. A. Bonnaud, B. Coasne, and R. J.-M. Pellenq. "Molecular simulation of water confined in nanoporous silica". In: *Journal of physics. Condensed matter: an Institute of Physics journal* 22.28 (2010), p. 284110.

[6] A. Carbone. "Algorithm to estimate the Hurst exponent of high-dimensional fractals". In: *Physical Review E* 76.5 (2007), p. 056703.

[7] A. G. Császár et al. "On equilibrium structures of the water molecule." In: *The Journal of chemical physics* 122.21 (2005), p. 214305.

[8] J. R. Errington and P. G. Debenedetti. "Relationships between structural order and the anomalies of liquid water". In: *Nature* 409.January (2001), pp. 318–321.

[9] J. Fan. "Rescaled Range Analysis in Higher Dimensions". In: *Research Journal of Applied Sciences, Engineering and Technology* 5.18 (2013), pp. 4489–4492.

[10] J. Feder. *Fractals*. Springer, 1988.

[11] A. Fournier, D. Fussell, and L. Carpenter. "Computer rendering of stochastic models". In: *Communications of the ACM* 25.6 (1982), pp. 371–384.

[12] *FRACLAB 2.1, A Fractal Analysis Toolbox for Signal and Image Processing*. URL: http://fraclab.saclay.inria.fr/ (visited on 01/20/2014).

[13] D. Frenkel and B. Smit. *Understanding Molecular Simulation*. 2nd Edition. Academic Press, Inc., 2001.

[14] M. Griebel, S. Knapek, and G. Zumbusch. *Numerical Simulation in Molecular Dynamics: Numerics, Algorithms, Parallelization, Applications*. Springer, 2007.

[15] G.-F. Gu and W.-X. Zhou. "Detrending moving average algorithm for multifractals." In: *Physical Review E* 82.1 (2010), p. 011136.

[16] W. G. Hoover. "Canonical dynamics: equilibrium phase-space distributions". In: *Physical Review A* 31.3 (1985), pp. 1695–1697.

[17] W. G. Hoover. "Constant-pressure equations of motion". In: *Physical Review A* 34.3 (1986), pp. 4–5.

[18] H. E. Hurst. "Long-term Storage Capacity of Reservoirs". In: *American Society of Civil Engineers* 116 (1951), pp. 770–808.

[19] H. E. Hurst, R. P. Black, and Y. M. Simaika. *Long-Term Storage: An Experimental Study*. Constable, 1965.

[20] P. Kumar, S. V. Buldyrev, and H. E. Stanley. "A tetrahedral entropy for water." In: *Proceedings of the National Academy of Sciences of the United States of America* 106.52 (2009), pp. 22130–4.

[21] J. E. Lennard-Jones. "On the Determination of Molecular Fields. II. From the Equation of State of a Gas". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 106.738 (1924), pp. 463–477.

[22] N. Levinson. "The Wiener RMS (root mean square) error criterion in filter design and prediction". In: *MIT Journal of Mathematics and Physics* 25 (1947), pp. 261–278.

[23] P. D. Lickiss. "The synthesis and structure of organosilanols". In: *Advances in Inorganic Chemistry* 42 (1995), pp. 147–262.

[24] C. A. Mack. "Fifty Years of Moore's Law". In: *IEEE Transactions on Semiconductor Manufacturing* 24.2 (2011), pp. 202–207.

[25] B. Mandelbrot. *The Fractal Geometry of Nature*. 1983.

[26] B. B. Mandelbrot. "Technical Correspondence: Comment on Computer Rendering of Fractal Stochastic Models". In: *Communications of the ACM* 25.8 (1982), pp. 581–583.

[27] B. B. Mandelbrot and J. W. V. Ness. "Fractional Brownian Motions, Fractional Noises and Applications". In: *SIAM Review* 10.4 (1968), pp. 422–437.

[28] G. J. Martyna, M. L. Klein, and M. Tuckerman. "Nosé–Hoover chains: The canonical ensemble via continuous dynamics". In: *The Journal of Chemical Physics* 97.4 (1992), p. 2635.

[29] G. J. Martyna et al. "Explicit reversible integrators for extended systems dynamics". In: *Molecular Physics* 87.5 (1996), pp. 1117–1157.

[30] G. E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965).

[31] S. Nosé. "A unified formulation of the constant temperature molecular dynamics methods". In: *The Journal of Chemical Physics* 81.1 (1984), p. 511.

[32] Persistence of Vision Pty. Ltd. (2004). *Persistence of Vision (TM) Raytracer [Computer software]*. URL: http://www.povray.org/ (visited on 05/21/2014).

[33] F. Renard, T. Candela, and E. Bouchaud. "Constant dimensionality of fault roughness from the scale of micro-fractures to the scale of continents". In: *Geophysical Research Letters* 40.1 (2013), pp. 83–87.

[34] C. H. Rycroft. *Voro++ [Computer software]*. URL: http://math.lbl.gov/voro++/ (visited on 05/21/2014).

[35] C. H. Rycroft. "VORO++: A three-dimensional voronoi cell library in C++". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 19.4 (2009), p. 041111.

[36] D. Saupe. "Algorithms for Random Fractals". In: *The Science of Fractal Images*. Ed. by H.-O. Peitgen and D. Saupe. Springer-Verlag New York, Inc., 1988. Chap. 2, pp. 71–113.

[37] Y.-H. Shao et al. "Comparing the performance of FA, DFA and DMA using different synthetic long-range correlated time series." In: *Scientific reports* 2 (2012), p. 835.

[38] A. Shekhar et al. "Nanobubble Collapse on a Silica Surface in Water: Billion-Atom Reactive Molecular Dynamics Simulations". In: *Physical Review Letters* 111.18 (2013), p. 184503.

[39] A. Shekhar et al. "Supplemental Material to Nanobubble Collapse on a Silica Surface in Water". In: *Physical Review Letters* 111.18 (2013).

[40] A. Stukowski. "Visualization and analysis of atomistic simulation data with OVITO–the Open Visualization Tool". In: *Modelling and Simulation in Materials Science and Engineering* 18.1 (2010), p. 015012.

[41] W. C. Swope. "A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: Application to small water clusters". In: *The Journal of Chemical Physics* 76.1 (1982), p. 637.

[42] The Inkscape Team. *Inkscape [Computer software]*. URL: http://www.inkscape.org/ (visited on 05/26/2014).

[43] M. E. Tuckerman, B. J. Berne, and G. J. Martyna. "Reversible multiple time scale molecular dynamics". In: *The Journal of Chemical Physics* 97.3 (1992), p. 1990.

[44] P. Vashishta et al. "Interaction potential for SiO2: A molecular-dynamics study of structural correlations". In: *Physical Review B* 41.17 (1990), pp. 12197–12209.

[45] L. Verlet. "Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Physical Review* 159.1 (1967), pp. 98–103.

[46] R. F. Voss. "Random Fractal Forgeries". In: *Fundamental Algorithms for Computer Graphics*. Ed. by R. A. Earnshaw. Vol. 17. Proceedings of the NATO Advanced Study Institute on Fundamental Algorithms for Computer Graphics held at Likley, Yorkshire, England, March 30 - April 12, 1985. See also [47]. Springer-Verlag, 1985. Chap. 8, pp. 805–835.

[47] R. F. Voss. "Fractals in nature: From characterization to simulation". In: *The Science of Fractal Images*. Ed. by H.-O. Peitgen and D. Saupe. See also [46]. Springer-Verlag New York, Inc., 1988. Chap. 1, pp. 21–70.

[48] Wikimedia Commons. *Voronoi*. 6, 2013. URL: http://en.wikipedia.org/wiki/File:Euclidean_Voronoi_Diagram.png (visited on 05/21/2014).

[49] A. T. A. Wood and G. Chan. "Simulation of stationary Gaussian processes in [0,1]^d". In: *Journal of computational and Graphical Statistics* 3.4 (1994), pp. 409–432.

[50] G. Zhou and N. S.-N. Lam. "A comparison of fractal dimension estimators based on multiple surface generation algorithms". In: *Computers & Geosciences* 31.10 (2005), pp. 1260–1269.

[51] L. Zhuravlev. "The surface chemistry of amorphous silica. Zhuravlev model". In: *Colloids and Surfaces A: Physicochemical and Engineering Aspects* 173.1-3 (2000), pp. 1–38.