Brownian Dynamics Simulation of Migration of DNA in structured Microchannels

Dissertation

zur Erlangung des Doktorgrades

vorgelegt von

Martin Andreas Streek
Theorie der kondensierten Materie
Fakultät für Physik
Universität Bielefeld

begutachtet durch

Prof. Dr. Friederike Schmid
PD Dr. Robert Ros

vorgelegt am

7. Januar 2005

Contents

1 Introduction

In the last years, scientific progress in biotechnology has been enormous. Much of the research effort has been spent on DeoxyriboNucleic Acid (DNA). DNA is one of the most important molecules in the cell, as it contains the genetic code. For example, it describes the sequences of animo acids of the proteins, and the structure of the organism as well.

The importance of DNA in the cell has led to much research effort being spent on the properties of this molecule. Today, many applications are available, which did not exist before:

- For example, it is now possible to identify people by their genetic fingerprint, which is already a standard application in forensic research. In many cases, a unique identification is still possible after years.
- Another application is the investigation of genealogies. Here, genetic fingerprints of different persons are investigated. If the patterns show identical parts, the persons are relatives of each other.
- By sequencing (*i. e.* analyzing the sequence of base pairs, Ch. 1.1) certain parts of the DNA, it is possible to detect hereditary diseases even before the fetus is born.
- Another possibility is to inject a DNA molecule into a stem cell, or even a single egg cell. This leads to an identical clone of a body part, or a complete new organism with the same genes as the original one.

Note that all these applications have many benefits on the one hand. On the other hand, all these applications may easily lead to problems. For example, if a prenatal investigation indicates that the fetus will suffer from a hereditary disease at the age of 20, the decision whether to perform an abortion or not is a severe decision. In this case, no investigation at all would have also been a considerable option. This example shows that a thorough discussion about applications in biotechnology is needed, and already performed.

I am convinced that acceptable solutions will be found, and will not discuss possible applications like the ones described above. The applications above rely on a set of efficient tools to handle DNA. One of these tools is an efficient separation of DNA fragments by length (without breaking the DNA), which I will investigate in this thesis. An overview of the chemical structure of DNA is given in chapter 1.1, and current separation techniques are introduced in chapter 1.2. For simulation, I used Brownian dynamics of a coarse-grained model of DNA. The simulation model is discussed in chapter 2. I performed simulations on entropic trap arrays with a DC field (Ch. 3), on geometrically structured micro channels with a DC field (Ch. 4), and an AC field on an array related to the microchannel (Ch. 5). This thesis is concluded in chapter 6.

1.1 Structure of double-stranded DNA

Double stranded DNA (ds-DNA) consists of two complementary strands. Each of these consists of a backbone, to which the nucleic acids are connected. The most common nucleic acids are shown in figure 1.1. However, unusual acids also exist [1]. The acids

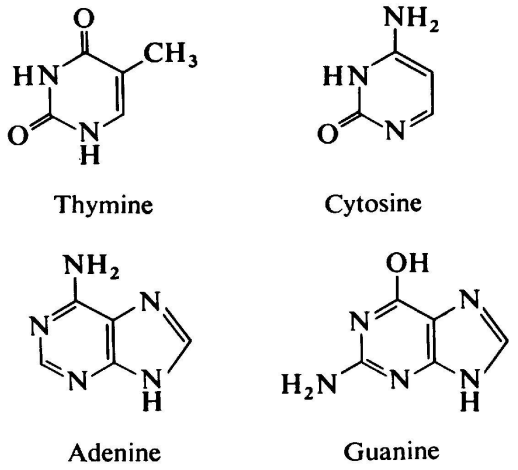

Figure 1.1: Acid bases of DNA [1]

are connected to a backbone, which is a periodic sequence of phosphoric acids and sugars. In the double stranded form, two opposing nucleic acids form one base pair. The opposing base pairs are:

- (T)hymine ↔ (A)denine
- (C)ytosine ↔ (G)uanine

Base pairs consisting of Thymine and Adenine are connected by two hydrogen bonds, and Cytosine and Guanine are attached to each other by three hydrogen bonds [2]. This pairing was discovered by Watson and Crick in 1953 [3]. The structure of double stranded DNA is given in figure 1.2. As the two types of opposite base pairs occupy roughly the same shape, the folding of double stranded DNA can be regarded as independent of the base pair sequence.

These two strands form a double helix, with the sugar-phosphate backbone on the outside, and the basepairs inside the double helix. Note that each single base pair is located off the helix axis. Thus the two strands are not located symmetrically around the helix axis, and the two strands can be distinguished from each other. DNA replication is a rather complex procedure, as the two strands are not equal in the double helix, and the backbone also defines a reading direction. Therefore, the replication process works periodically in different stages. A description is given in references [4, 5].

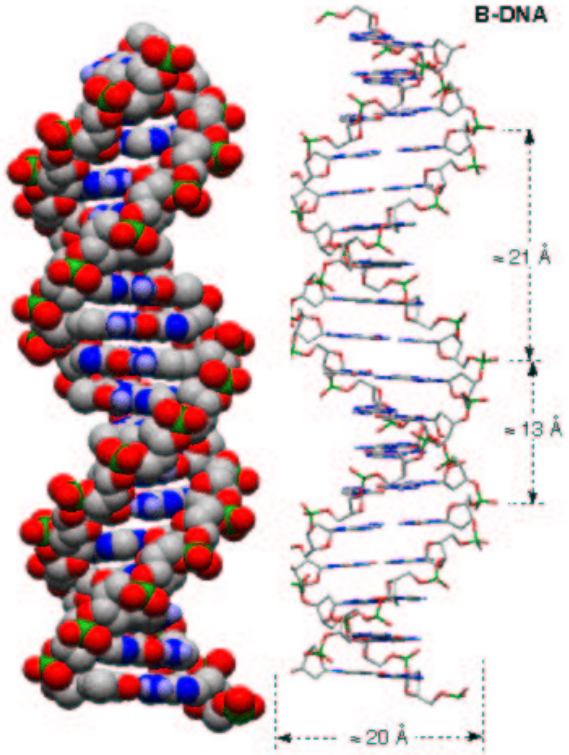

Figure 1.2: Structure of double stranded DNA, shown by calotte (left) and rod model (right). Complemenatry base pairs are located inside the two sugar-phosphate backbones. Taken from
`http://www.oci.unizh.ch/edu/lectures/material/AC_BII`.

1.1.1 Properties of double-stranded DNA

When comparing simulation to experimental data, it is necessary to match the relevant parameters of the experiment to the corresponding ones in simulation. The most relevant properties of double stranded DNA, combined with their consequences to computer simulations will be discussed below.

- DNA is a charged polyelectrolyte with two negative charges per base pair. Thus each monomer is subject to and electric field, and the force on a single monomer depends on the monomer's position only.

- DNA molecules are usually investigated in aqueous solution. This implies that the DNA fragment is surrounded by counter ions, which screen the charges of the DNA molecule. This layer can be separated into the Stern layer, which is fixed to the ion, and the diffuse layer, covering the Stern layer. In physiological condidtions, the thickness of the Stern layer is roughly given by the Bjerrum length, which is typically around 0.7nm in water, and the Debye length, on which electro-

static interactions are screened, is around 2nm. A detailed discussion is given in reference [6].

- As the base pairs occupy roughly the same space inside the double helix, the folding of double stranded DNA is independent on the sequence of base pairs. Thus even a coarse-grained DNA molecule is easy to describe. Note that this is much easier than, for example, protein folding [7], or even RNA folding [8].
- The radius of gyration R_g is typically around a few μm, depending on the length of the fragment. For example, λ-DNA consists of 48kbp, and exhibits a radius of gyration $R_g \approx 0.7\mu$m [9].
- The diameter of the DNA double helix is around 2nm, and thus comparable to the Debye screening length.
- DNA is a rather stiff polymer. In water, the persistence length is typically around 45nm [10, 11]. Thus electrostatic interactions along the polymer backbone can be neglected, as these are much shorter in range.
- The distance between two base pairs along the double helix is approximately 3.4Å, and ten base pairs form one full turn of the double helix [1]. Thus each turn of the helix has a length of 34Å.
- The number of base pairs stored in the DNA molecule reaches up to $3 \cdot 10^9$ for human DNA. Note that this amount of DNA is not stored in one strand, but in 46 chromosomes [1]. Assuming that the DNA fragments are equal in length, human DNA has a contour length of roughly 2cm.

To summarize, DNA is very interesting, not only for biologists, but for physicists as well, as it is the ideal polyelectrolyte. Electrostatics along the backbone do not contribute notably, as the double helix is rather stiff, but the application of an external force is simple. Treating hydrodynamics of polymers in solution is, in general, a delicate task [12]. In this case, however, electroosmotic and hydrodynamic interactions screen each other [6]. Furthermore, the folding is independent of the sequence of base pairs, allowing for a simple model. Thus it is possible to model double stranded DNA on a coarse-grained level by a stiff bead-spring model. The model used in simulation is described in chapter 2.

1.2 Separation Devices

Currently, many different separation devices are in use or under development [13, 6]. As DNA is a charged polyelectrolyte, separation is usually achieved by a subtle interplay of an applied electric field and geometric barriers of some kind.

One type of device uses a polymer gel as a sieving matrix. Here, the entangled polymer strands form a random sieving matrix. Electrophoresis in polymer gels is widely used and well understood.

Another category of separation devices is based on artificial sieving matrices. An overview of these μ – TAS (Total Analysis Systems) devices and their biological applications is given in reference [14]. These devices benefit from the possibility of being designed for a special purpose, even sorting of cells is possible with a cheap device [15]. Whereas in gels the pore size varies and is statistically distributed, it is easy to construct devices very accurately due to the recent advances in micro technology [16]. Furthermore, it is possible to exploit interactions with the walls, as the surface to volume ratio cannot be ignored [17]. In small microfluidic channels, the migration of long DNA molecules is already affected when the walls are smooth and chemically neutral [18]. A thorough examination of polyelectrolytes in solution and at walls by computer simulations is given in reference [19]. Is is also possible to chemically modify the surface of the device for a special purpose, like antibodies, polymer brushes, etc. [20]. In addition, the relaxation times need to be considered while constructing a device, as it may easily exceed the migration time from one obstacle to another [21]. This is also discussed in chapter 3.4.2 and reference [22]. As the cross section of the device is very small, it is now possible to observe single molecules by video microscopy during migration [23]. With a proper geometry and a pulsed field, it is also possible to mix sample volumes [24]. Mixing is also possible by hydrodynamic focusing [25], or by a simple T-form device [26]. An overview of some mixing techniques is given in references [27, 28]. Note that if the hydrodynamic flow exceeds a critical value, DNA may fragment [29]. In that case, the device is not useful for separation, as it destroys the sample DNA.

A few of these separation devices shall be described below. Note that separation of DNA by electrophoresis in free solution does not work, as DNA exhibits a friction that proportional to the number of base pairs, which just balances out the effect of the homogeneous charge along its own contour [6].

1.2.1 Electrophoresis in a Gel

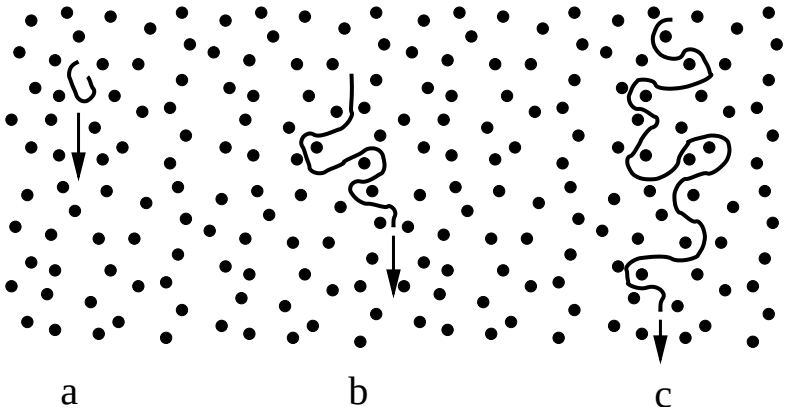

Figure 1.3: Electrophoresis in a gel: Ogston sieving (a), reptation without orientation (b), and biased reptation (c)

Up to recent days, electrophoresis is often performed in polymer gels. A schematic drawing of the different regimes is given in figure 1.3:

a: Very short DNA fragments with a diameter comparable to the pore size have to squeeze through the pores of the gel as a whole. This regime is called Ogston sieving.

b: At a certain length, the chains are no longer able to squeeze through a single pore without large deformations. In this case, one loop of the DNA fragment will penetrate a pore, and the rest will get pulled through that pore by the initial loop. This regime is called "reptation without orientation". The name "reptation" is related to the movement of the DNA strand, which is very similar to that of a snake moving between obstacles [30, 31, 32].

c: Long DNA fragments reveal a migration behavior which is very similar to that described in (b), but in this case the DNA fragment is sufficiently long to exhibit an orientation. This regime is called "biased reptation".

Electrophoresis in a polymer gel has the advantage that it is widely used and well understood [6]. However, the mobility μ of a DNA fragment with N base pairs in the biased reptation regime is given by:

$$\left(\frac{\mu}{\mu_0}\right)_{\mathrm{BRM}} \approx \frac{1}{3}\left(\frac{1}{N} + \frac{\epsilon^2}{3}\right), \tag{1.1}$$

where μ_0 is the free-flow mobility, and ϵ represents the reduced electric field [6]. As a consequence, electrophoresis in a gel is unable to efficiently separate long DNA strands [33, 34]. In experiments, separation becomes inefficient for DNA strands longer than $\sim$ 20kbp [35], and fragments longer than $\sim$ 40kbp cannot be separated any more [36].

One way to enhance the separation capability of electrophoresis in gels is to use pulsed electric fields [37], which employ the direction of the DNA molecule in biased reptation: the reorientation of the molecule direction does depend on the fragment's length. Successful separation of up to 10 Mbp has been reported [38, 39]. Unfortunately, the separation in pulsed field gel electrophoresis needs the DNA fragments to relax. Therefore, separation usually takes from many hours up to several days [40].

1.2.2 Capillary Gel Electrophoresis

One way to effectively enhance the separation performance of gels is to use gel electrophoresis in a capillary [41]. Whereas the migration mechanisms of DNA in gels are mostly unaffected, additional effects arise from the fact that the DNA fragments are now interacting with the walls of the capillary, whose diameter is usually around $50 - 100\mu$m [6]. Furthermore, it is possible to enhance the separation performance in an array [13], and it is possible to use it in a lab-on-a-chip device [16]. Books on capillary electrophoresis are given in references [42, 43, 44].

Capillary gel electrophoresis has also been investigated by computer simulations. Dose *et al.* [45] have computed the development of flows during the separation, and Mosher

et al. [46] have presented a model to describe electrophoresis in capillaries including electroosmotic flow.

The biggest success for gel electrophoresis in capillaries was the successful sequencing of the human genome in the year 2000, years ahead of the original estimate [47, 48].

However, it is not trivial to incorporate a polymer gel into a capillary, especially those used in ordinary gel electrophoresis. The polymers may break, bubbles can occur, or the device gets clogged in the production process [6]. On the other hand, it is possible to use fluids, which are not usable in free-flow electrophoresis.

1.2.3 Entropic Trap Arrays

Another length-dependent mechanism has been discovered by Baumgärtner and Muthukumar [49, 50, 51]. It is different from the Ogston sieving and reptation described above (Ch. 1.2.1, Ref. [52]). A schematic drawing of entropic trapping is shown in figure 1.4.

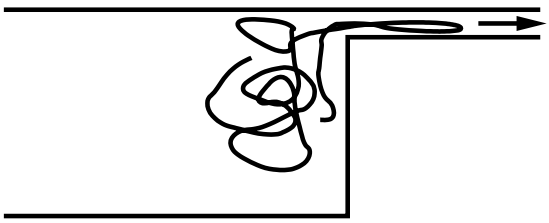

Figure 1.4: Entropic trapping: The penetration of the narrow region by the initial loop reduces the entropic free energy of the coil.

Here, the polymer gets stuck at a sudden narrowing of the device, and an initial loop penetrates the narrow region. This reduces the entropic free energy of the polymer, hence the name. As the entropic energy loss increases with the chain length, long chains are naively expected to migrate slower than short ones.

Han and Craighead have successfully constructed an entropic trap device, and found length-dependent behavior due to entropic trapping [53, 54, 55, 56]. In their experiments, the width of the narrow region is around 90nm, which is the same order of magnitude as the persistence length of DNA. They also performed a simple analytic calculation, and are able to explain the counter-intuitive finding that long chains migrate faster than short ones.

These results have motivated several computer simulations. A small selection of these is described below.

A simulation of the device presented by Han *et al.* has been performed by Tessier *et al.* [57] by a Monte Carlo simulation on a lattice. The simulation results confirm the findings of Han *et al.*. Later, an off-lattice, Brownian dynamics simulation was performed by me (Ch. 3), and a new trapping mechanism discovered (Ch. 3.4.5), in collaboration with Friederike Schmid [58].

An examination of the free-energy landscape of the escape process was performed by Chen *et al.* [59]. The findings do not fully support the results found by Han *et al.*.

However, one possible explanation for the deviations can be found in the fact that long chains are unable to relax into a coiled conformation, as is shown in chapter 3.4.2. Therefore, back-to-back escape processes are correlated.

Tessier *et al.* [60] have also investigated the influence of a pulsed electric field with mean 0. They find that entropic trapping can be exploited as a ratchet mechanism.

1.2.4 Geometrically structured Microchannels

Doung *et al.* have presented a device which is very similar to entropic traps [61, 62]. A schematic drawing is shown in figure 1.5, and an SEM picture of the actual device is shown in figure 4.1. It also consists of a periodic sequence of wide and narrow regions,

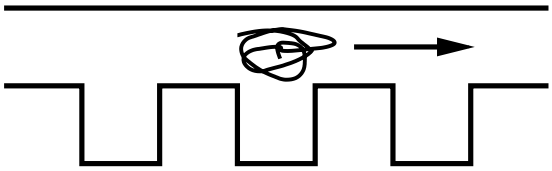

Figure 1.5: Geometrically structured microchannel. The width of the narrow region allows for a coiled conformation of the polymer.

but the width of the narrow regions is comparable to the radius of gyration of the DNA coil. Thus no uncoiling is necessary when passing the narrow region, and an entropic trapping mechanism cannot be applied here. However, chains may still get caught in the wide region, as described in chapter 3.4.5, and long chains were expected to migrate faster than short ones.

In experiments, the opposite migration order was observed, *i. e.* long DNA fragments migrated slower than short ones [61, 62]. At even higher fields, two distinct migration states are observed [63, 64]. This unforeseen behavior is discussed in detail in chapter 4.

1.2.5 Microsieves

Many more separation devices have been constructed. A brief overview of some is given below.

Nixon *et al.* have investigated entropic trapping in channels with periodically varying width [65]. In particular, they investigated two critical field strengths: the minimum field that is necessary to overcome the entropic free energy barrier and also the field for which the longitudinal diffusion reaches a maximum. In their simulations, they find three regimes: at low fields, entropic trapping dominates the migration, and at high fields the migration is not influenced by entropic trapping, but by an effective friction due to the barriers. At intermediate fields, both these effects affect the migration.

Slater *et al.* have shown that asymmetric cavities are able to induce a net flow for zero-mean, time-symmetric electric fields [66]. They also find that entropic trapping is able to enhance ratchet effects.

It is also possible to exploit a complex flow pattern. In such a device, it is possible to stretch DNA molecules by a hydrodynamic flow at channel junctions [67], or by an induced shear [68].

Curves in the microchannel may also be used as an efficient separation device [69, 70]. Such a device is even capable of a separation in half a minute. DNA molecules may also be sorted in a simple T-junction device [71], and this device does not depend on the mobility of the sample DNA. Separation of a DNA-ladder from $2 - 200$kbp is reported.

Turner *et al.* have presented length dependent mobility in an artificial gel consisting of an rectangular array of sub-micron obstacles [72]. They reported successful separation of the investigated DNA strands, and the mobilities differed by a factor of almost 2. A very similar device has been presented by Bakajin *et al.* [35]. Here, the device consists of a hexagonal array of pillars with 2μm width. Separation of 100kbp-DNA is reported in ~ 10 seconds with an asymmetrically pulsed electric field. Later, separation of DNA fragments with $61 - 209$kbp were separated in 15 seconds [40].

1.2.6 Ratchets

Ratchet effects can be used to induce a net flow with a mean zero field [73], and a few examples have already been described above. Basically, a ratchet is driven periodically. Furthermore, it is necessary to break the symmetry in either the time dependence of the applied field, or the setup of the device. An example showing a ratchet mechanism due

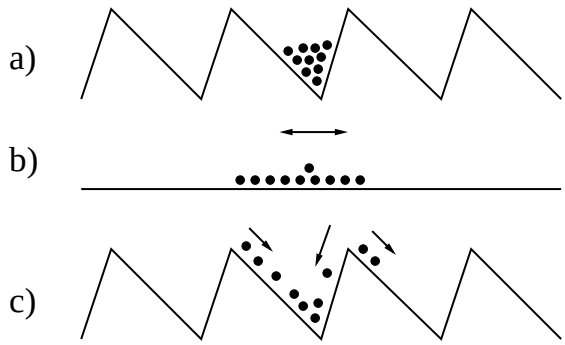

Figure 1.6: Ratchet effect due to an asymmetric electric field. When the field is switched on, all particles move towards the minima of the potential (a). At some time, the electric field is switched off, the particles can diffuse freely (b). Afterwards, the electric field is switched on again. As the peaks of the potential are not symmetric around the minima, a net flow to the right is induced (c). For a better understanding, only one minimum is filled in the initial state.

to broken symmetry of the space is shown in figure 1.6. In this case, an electric field is switched on and off periodically. When the electric field is switched on, all particles move towards the minima of the electric field (a). Later, the electric field is switched off, and the particles may diffuse freely in both directions (b). At this point, the distribution

of the particles can be assumed Gaussian, the center located at the minimum. When the field is switched on again, the particles move according to the electric field. As the distances to the peaks surrounding the initial minimum differ on both sides, more particles have diffused sufficiently to move to the right than to the left. Thus, a net drift to the right is induced. Successful driving of DNA by this mechanism is reported in reference [74, 75].

It is also possible to use a ratchet combined with a force gradient (pressure, gravity, etc). In that case, a continuous operation of the device is possible, and particles below a certain threshold move to the one side, and above the threshold to the other [76]. It is also possible to combine a polymer gel with a ratchet mechanism [77]. Another way to exploit a ratchet mechanism is to use a pulsed electric field with mean zero (Fig. 2.9), or with an asymmetric geometry [78, 79].

2 The Simulation Model

There are two major techniques in computer simulation. On the one hand side, Monte Carlo (MC) simulations compute the partition function by proposing single moves which are either rejected or accepted according to a "Metropolis" criterion [80]. Although this method is very fast and efficient, it is not useful to study time-dependent or non-equilibrium systems. In the work described here, the system is far from equilibrium due to the external electric field.

The other common method is called molecular dynamics (MD) simulation [80, 81]. In this case, the equations of motion are integrated numerically:

$$m_i \ddot{\vec{r}}_i = -\left.\frac{\partial V}{\partial \vec{r}}\right|_{\vec{r}=\vec{r}_i} \tag{2.1}$$

MD simulation requires careful selection of the integration algorithm. First, the computation of the forces $f = -\frac{\partial V}{\partial \vec{r}}$ may be complex. Thus a sophisticated algorithm has to work properly with one evaluation of the forces per time step only. Moreover, it is important that the algorithm provides accurate results:

- Short-time accuracy:
  The integration algorithm has to be accurate in a single, finite time step. The time step is favorably rather large to speed up simulations.

- Long-time stability:
  It is most important that the algorithm is stable on long time scales, *i. e.*, it conserves conservation laws and symmetries. For example, in the micro canonical ensemble, the energy of the system has to be conserved:
  $\lim_{t\to\infty} \left|\frac{E(t)-E(0)}{E(0)}\right| \ll 1$

2.1 Integration Algorithms

2.1.1 Verlet Algorithm

The usual integration algorithm for MD simulation is the Verlet algorithm. It is based on the following time step:

$$\vec{r}_i(t+\Delta_t) = 2\vec{r}_i(t) - \vec{r}_i(t-\Delta_t) + \frac{h^2}{m_i}\vec{f}_i(t) + \mathcal{O}(\Delta_t^4) \tag{2.2}$$

$$\vec{v}_i(t) = \frac{1}{2\Delta_t}\left[\vec{r}_i(t+\Delta_t) - \vec{r}_i(t-\Delta_t)\right] + \mathcal{O}(\Delta_t^2), \tag{2.3}$$

which can be derived from the Taylor series of the real trajectory. In this form, it is inconvenient for computer simulation. There is a subtraction of two large numbers in equation 2.2, when the difference $2\vec{r}_i(t) - \vec{r}_i(t - \Delta_t)$ is computed. This may lead to large numeric errors in simulation. Furthermore, the velocities can only be computed after one additional time step. However, it is possible to convert these equations to:

$$\vec{r}_i(t + \Delta_t) = \vec{r}_i(t) + h\vec{v}_i(t) + \frac{\Delta_t^2}{2m_i}\vec{f}_i(t) + \mathcal{O}(\Delta_t^3) \tag{2.4}$$

$$\vec{v}_i(t + \Delta_t) = \vec{v}_i(t) + \frac{\Delta_t}{2m_i}\left(\vec{f}_i(t) + \vec{f}_i(t + \Delta_t)\right) + \mathcal{O}(\Delta_t^3) \tag{2.5}$$

These equations are algebraically equivalent to the equations 2.2 and 2.3, and are known as the Velocity-Verlet algorithm. It has the same advantages as the Verlet algorithm, which are:

- Local error of order Δ_t^3
- Global error of order Δ_t^2
- The two equations are symmetric in Δ_t, hence the algorithm is time-reversible
- Exact phase-space conservation ("symplectic")

Benefiting from the properties described above, the Velocity-Verlet algorithm has good short-time accuracy and excellent long-time stability. Today, the Verlet algorithm is standard in MD simulation [80].

2.1.2 Overdamped Dynamics

Although the Verlet algorithm provides excellent properties for MD simulation, sometimes it is necessary to perform simulations in the limit $m \to 0$. Thus a definition of the velocity is only possible by $\vec{v}_i(t) \approx \frac{\vec{r}_i(t+\Delta_t) - \vec{r}_i(t)}{\Delta_t}$. A very simple algorithm that computes the forces only once per time step and is capable of integrating overdamped systems is the Euler algorithm. It is based on the time step

$$\vec{r}_i(t + \Delta_t) = \vec{r}_i(t) + \Delta_t \cdot \vec{f}_i(t). \tag{2.6}$$

The properties of this algorithm are rather poor:

- Local error of order Δ_t^2
- Global error of order Δ_t
- The algorithm is not time-reversible
- The algorithm is not symplectic

However, in the case of Langevin dynamics (Ch. 2.2), time-reversibility and symplecticity are destroyed by the friction and the random force. This also destroys any higher-order convergence of the error. Thus the Euler algorithm is suitable for simulating overdamped Langevin dynamics.

2.2 Langevin Dynamics

On the one hand, the interaction of the DNA molecule with the surrounding solvent particles cannot be neglected during simulation. On the other hand, it is impossible to incorporate a full-scale MD simulation, because in the limit of long polymers the monomer density inside the coiled polymer goes to zero [82]. Thus the solvent particles need to be replaced by an external interaction describing the impulse transfered by the solvent molecules.

In this case, the effect of the solvent molecules is described by a Langevin equation (Eq. 2.7 and 2.8). The system is described by a Kramer's equation [83]: the solvent is replaced by an effective friction ζ and a random force $\vec{\eta}$. Thus the equation of motion for particle i becomes

$$\dot{\vec{r}}_i = \vec{v}_i \tag{2.7}$$

$$m\dot{\vec{v}}_i = \vec{f}_i - \zeta\vec{v}_i + \vec{\eta}_i, \tag{2.8}$$

where $\vec{r}_i$ is the position, $\vec{v}_i$ is the velocity and $\vec{f}_i$ is the deterministic force acting on the particle. This kind of dynamics is also called Brownian dynamics, as it describes Brownian motion of a particle on a coarse-grained level.

Given the friction ζ, the random force $\vec{\eta}_i$ has to fulfill [84]

$$< \vec{\eta}_i > = 0 \tag{2.9}$$

$$< \eta_{i,\alpha}(t)\eta_{j,\beta}(t') > = 2\zeta k_{\mathrm{B}}T\delta_{i,j}\delta_{\alpha,\beta}\delta(t-t') \tag{2.10}$$

with cartesian coordinates $\{\alpha, \beta\} = \{x, y, z\}$, and t and t' two given times. Equation 2.9 simply demands that the random force is unbiased and equation 2.10 demands that the random force has to be uncorrelated for different particles, cartesian coordinates and different times. Furthermore, it defines the variance of the random force. For the particle i, this can be integrated to the well-known diffusion equation

$$\left\langle (\vec{r}_i(t) - \vec{r}_i(0))^2 \right\rangle = 6\frac{k_B T}{\zeta}t =: 6Dt \tag{2.11}$$

with the diffusion constant D.

2.2.1 Langevin Dynamics in the Verlet Algorithm

Substitution of the Langevin equation into the Verlet algorithm leads to the following time step:

$$\begin{aligned}
\vec{v}_i(t+\tfrac{\Delta_t}{2}) &= \vec{v}_i(t) + \tfrac{\Delta_t}{2m_i}\vec{f}_i(\vec{r}_i(t)) - \tfrac{\zeta}{m_i}\Delta_t\vec{v}_i(t) + \tfrac{1}{m_i}\sqrt{2k_{\mathrm{B}}T\zeta\Delta_t}\hat{\eta}_i(t) \\
\vec{r}_i(t+\Delta_t) &= \vec{r}_i(t) + \Delta_t\vec{v}_i(t+\tfrac{\Delta_t}{2}) \\
\vec{v}_i(t+\Delta_t) &= \vec{v}_i(t+\tfrac{\Delta_t}{2m_i}) + \tfrac{\Delta_t}{2}\vec{f}_i(\vec{r}_i(t+\Delta_t))
\end{aligned} \tag{2.12}$$

The random number $\hat{\eta}_i$ is proportional to $\vec{\eta}_i$ and fulfills

$$< (\hat{\eta}_{i,\alpha})^2 >= 1. \tag{2.13}$$

The random numbers $\vec{\eta}_i(t)$ do not have to be Gaussian, equally distributed random numbers are sufficient as well [85]. In the simulation, the random numbers $\hat{\eta}_i$ are distributed equally inside the unit sphere. These are computed much more easily [86].

Selection of the Time Step

To compute the short-time dynamics correctly, it is necessary to keep the time step low. However, long-time averages are affected by the time step as well, which has already been shown for a single particle [86]. In this case, the critical parameter of the time step is $\zeta\Delta_t$, which should be kept low.

2.2.2 Overdamped Dynamics

It turns out that the simulation model described above shows inertia effects, which in some cases is an artifact, as relaxation times may be prolonged compared to experiment 4.3. Therefore, some simulations were performed using overdamped dynamics to test the validity of the model.

In the case of $m \to 0$, the equations of motion ((Eq. 2.7 and 2.8) become

$$\zeta \dot{\vec{r}}_i = \vec{f}_i + \vec{\eta}_i. \tag{2.14}$$

These are substituted into the Euler algorithm described above (Eq. 2.6):

$$\vec{r}_i(t + \Delta_t) = \vec{r}_i(t) + \frac{1}{\zeta}\left[\Delta_t \vec{f}_i + \sqrt{2k_\mathrm{B}T\zeta\Delta_t}\hat{\eta}_i\right], \tag{2.15}$$

where $\hat{\eta}$ fulfills equation 2.13.

This algorithm has been used successful before [87].

Diffusion

The mean quadratic drift of a free particle in one time step can be computed easily:

$$< (\vec{r}_i(t + \Delta_t) - \vec{r}_i(t))^2 >= \frac{2k_\mathrm{B}T\zeta\Delta_t}{\zeta^2} < \hat{\eta}_i^2 >= \frac{6k_\mathrm{B}T}{\zeta}\Delta_t = 6D\Delta_t \tag{2.16}$$

Thus the equation of diffusion (Eq. 2.11) is fulfilled naturally in every time step.

2.3 The Simulation Model

The properties of a polymer of sufficient length are mostly dependent on the monomer number, the internal structure becomes neglectable [84]. The simplest polymer model is the Gaussian Chain Model (Fig. 2.1): in this model, neighboring monomers are connected

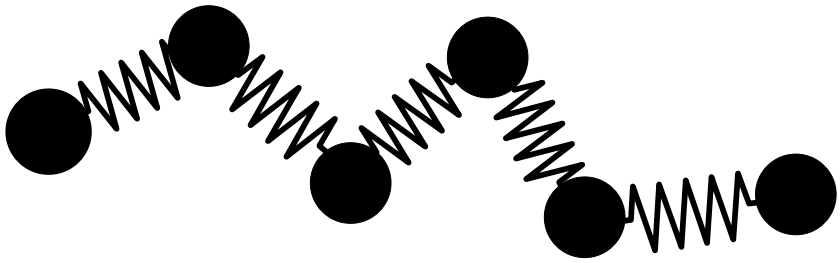

Figure 2.1: Gaussian chain model: Schematic drawing

by harmonic springs with spring constant k

$$V_{\text{bond}} = \frac{1}{2} k r_{\text{bond}}^2. \tag{2.17}$$

To model the excluded volume, a repulsive Weeks-Chandler-Andersen (WCA) potential acting between all monomers is introduced. This is simply a Lennard-Jones potential which has been cut off at the equilibrium point [88, 89, 81]:

$$V_{\text{pair}}(r)/k_B T = \begin{cases} (\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^6 + \frac{1}{4} & : \quad (\frac{r}{\sigma}) \leq 2^{1/6}, \\ 0 & : \quad \text{otherwise}, \end{cases} \tag{2.18}$$

Although the interaction is short ranged in position space, it is long ranged regarded as a backbone interaction. Thus both local and global folding are affected and the exponent ν changes to 0.588 ± 0.001 [90, 91]. The scaling has been tested in reference [86].

For simplicity, the mass m has been set to unity in the simulation. Proper behavior of the model can be achieved by adjusting the parameters of the potentials.

This model is integrated with Langevin dynamics and a Verlet or Euler algorithm as described above. A detailed discussion of the properties of Gaussian chains is given in reference [84, 86]. Models of this kind have been used successfully before [92, 93, 94, 95, 96].

2.4 Natural Units of the Simulation

When adapting the simulation data to experimental data, it is necessary to match the natural units of the simulation to those of the experiment. These are the length $\sigma_{\text{LJ}} \equiv \sigma$ of the Lennard-Jones potential (Eq. 2.18), the friction ζ, the charge per bead $|q|$, and the Temperature T.

The energy unit of the simulation is $\in= k_{\text{B}}T$, which corresponds to the thermal energy of $300 k_{\text{B}}K \approx 0.026\text{eV}$ in experiments at room temperature. The time unit of the simulation is given by $t_0 = \sigma\zeta^2/k_{\text{B}}T$, and the electric field unit is $E_0 = k_{\text{B}}T/\sigma = k_{\text{B}}T/\sigma|q|$. The free-flow mobility μ_0 is given by $\mu_0 = |q|/\zeta = \sigma/t_0 E_0$.

These shall be related to experimental values in the chapters 3.3 and 4.3.

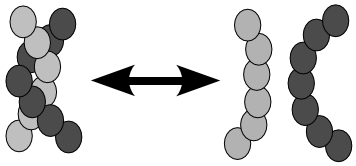

Figure 2.2: Entanglement: schematic drawing

2.5 Simulation Parameter Set

The mass has been set to unity during the simulation. The parameter values are:

$$T = 1 \tag{2.19}$$

$$\Delta_t = \begin{cases} 10^{-2} & : \text{ inertia dynamics} \\ 10^{-4} & : \text{ overdamped dynamics} \end{cases} \tag{2.20}$$

$$k_B = 1 \tag{2.21}$$

$$\zeta = 1 \tag{2.22}$$

$$\epsilon_{LJ} = 1 \tag{2.23}$$

$$\sigma = 1 \tag{2.24}$$

$$k = 100 \tag{2.25}$$

2.6 Entanglement

Another important effect in polymer dynamics is called entanglement. A schematic drawing is given in fig. 2.2. A sufficiently high barrier of the potential ensures that no bonds may cross each other. Usually, it is around $70k_{\mathrm{B}}T$, thus the Boltzmann factor for this event is neglectable [97]. In this simulation, the energy barrier is even higher, it is around $160k_{\mathrm{B}}T$. The bond length distribution agrees nicely with the prediction of the Boltzmann factor [86]. To further ensure no bond crossing occurs during simulation, the minimal and maximal bond length may be measured during simulation.

2.7 Properties of the free Chain

Most of the properties of the free chain have already been investigated in my diploma thesis. Therefore only a brief summary will be given below:

- The critical parameter of the dynamics with inertia is $\zeta\Delta_t = 0.01$, which is quite small. Therefore no correction of the temperature is necessary.
- Entanglement interactions (Ch. 2.6) are introduced by the tight springs. This has been checked before.

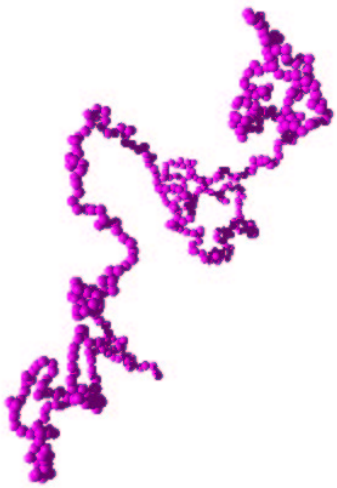

Figure 2.3: Snapshot of a free polymer [86]

- The mobility of a free chain can be computed via

$$\mu_0 = 1/\zeta \equiv 1$$

  and is constant for all chains.

- The average bond length is 0.854σ, the equilibrium bond length is $l_\mathrm{b} = 0.847\sigma$.

- The excluded volume also introduces a persistence length, which is around $l_\mathrm{p} \approx 1.6\sigma \approx 1.9 l_\mathrm{b}$.

- The diffusion constant D is given by

$$D = \frac{k_\mathrm{B}T}{N\zeta} \tag{2.26}$$

- For long chains, the radius of gyration can be approximated by

$$R_\mathrm{g} \approx 0.5 N^\nu, \tag{2.27}$$

  with N the number of monomers.

- Static properties of the chain, like the radius of gyration, R_g, or the persistence length, l_p, are independent of the dynamical model used in the simulation.

A snapshot of a polymer in free solution is given in figure 2.3.

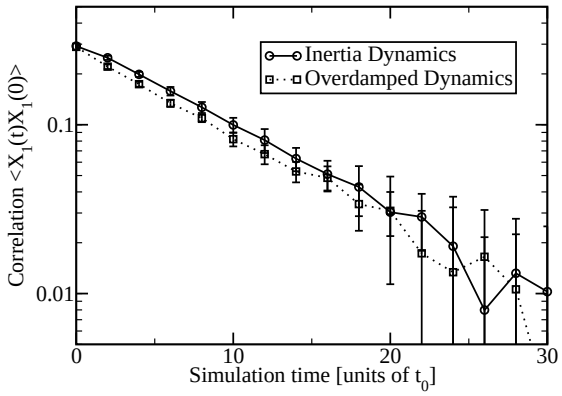

Figure 2.4: Relaxation of a polymer with $N = 10$ monomers. For short time scales ($t < t_0$), inertia affects the relaxation. Thereafter, the graphs agree very well.

2.7.1 Conformational Relaxation Time

One important property that has not been examined before for this parameter set is the conformational relaxation time. It is the de-correlation time τ_0 of the first Rouse mode

$$\vec{X}_1 = \frac{1}{N} \sum_n \left(\frac{(n + \frac{1}{2})\pi}{N} \right) \vec{r}_n \tag{2.28}$$

An example is given in figure 2.4. For short time scales (*i. e.* $t < t_0$), inertia effects dominate the decay of the correlation. On larger time scales, the agreement is very well. A sophisticated extraction of a single relaxation time is difficult, because the correlation on long time scales cannot be determined without huge errors. Therefore the values with large errors have been neglected in the analysis. As the determination of a meaningful cutoff is a delicate task for long chains, no errors were computed. The relaxation time has been determined for chain lengths up to $N = 100$. The results of this analysis are given in figure 2.5. The data for both dynamical models used agree very nicely, and can be fitted by a power law very well. Note that the value of $N = 100$ has been omitted in the analysis, as it deviates very strongly. The result of the regression is

$$\tau_{\mathrm{R}} = (0.048 \pm 0.005) N^{2.23 \pm 0.03}, \tag{2.29}$$

which agrees very well with the prediction $\tau_{\mathrm{R}} \propto N^{1+2\nu}$ [84, 98].

2.7.2 Time of Diffusion by R_{g}

Another important time scale in polymer dynamics is the the time it takes the polymer to diffuse by its own radius of gyration R_{g}. Using equation 2.11, this time τ_{D} can be easily evaluated:

$$\tau_{\mathrm{D}} = \frac{1}{6} D R_{\mathrm{g}}^2 = \frac{1}{6} \frac{N\zeta}{k_{\mathrm{B}}T} R_{\mathrm{g}}^2 \tag{2.30}$$

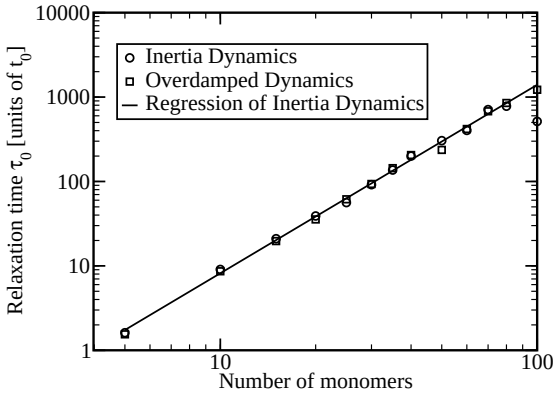

Figure 2.5: Relaxation time τ_{R} for various chain lengths. For long chains, the results deviate due to long relaxation times. Therefore the last value of the inertia dynamics has been omitted. The relaxation time from the regression is $\tau_{\mathrm{R}} = (0.048 \pm 0.005) N^{2.23 \pm 0.03}$.

The radius of gyration R_{g} has been computed before [86]. The radius of gyration was found to be $R_{\mathrm{g}} \approx 0.5 N^{\nu}$. Using equation 2.30, this leads to

$$\tau_{\mathrm{D}} \approx 0.042 N^{1+2\nu}, \tag{2.31}$$

which is very close to equation 2.29.

If the inertia of the chain is neglectable, *i. e.* a long chain or in the limit $m \to 0$, the conformational relaxation time τ_{R} can be estimated by evaluating the radius of gyration R_{g} and the diffusion constant D, which are easier to determine both in simulation as well as in experiment.

2.8 The Device

Generally, the device consists of a periodic sequence of thin and wide regions. A schematic drawing can be found in Fig. 2.6. Basically, four parameters are needed

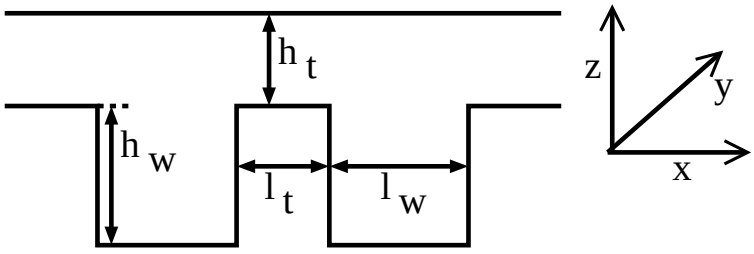

Figure 2.6: Schematic drawing of the investigated device. The coordinate system used in the simulations is shown on the right.

to describe the geometry of the device:

- the height of the wide region h_{w}
- the length of the wide region l_{w}
- the height of the thin region h_{t}
- the length of the thin region l_{t}.

Compared to my diploma work [86], tilted walls are no longer necessary because the electric field is no longer homogeneous (chapter 2.9). The walls are set up soft and purely repulsive. To describe the wall interaction, the Lennard-Jones potential that describes the monomer repulsion (Eq. 2.18) is used. This implies that the walls are covered with a repulsive layer of $\approx 1\sigma$. To compute the force exerted on a monomer, the nearest point of the wall is considered. In the corners, the forces of the adjacent walls are summed up.

For the case of entropic trap arrays [53, 54, 55, 56], the device is very large in y-direction. Thus no walls in that direction were introduced (Sec. 3). This idealization is no longer valid in the case of the device presented by Duong *et. al* [62]. The depth of the device is comparable to the radius of gyration of the DNA. To model the finite depth of the device, walls in y-direction were also introduced in the simulation (Sec. 4).

2.8.1 Parallelization of the Device

To achieve better separation performance, a parallelized version of the device has been simulated as well. A schematic drawing is given in figure 2.7. It consists of a periodic

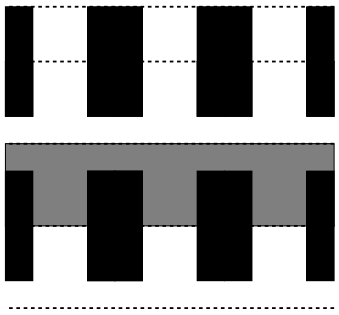

Figure 2.7: Schematic drawing of the parallelized device. The single device unit is shown in gray.

sequence of the device shown in fig. 2.6, which are alternatingly mirrored in the z-direction. The walls covering the device in z-direction have been removed. Due to the symmetry of this device, the electric field distribution inside the device is equivalent to that of a single device (chapter 2.9).

2.9 Electric Field

DNA is charged uniformly along its contour. Thus in simulation each bead carries the same charge, and is subject to an external force $\vec{F}_{\mathrm{el}} = \vec{E}(\vec{r}) \cdot q_{\mathrm{bead}}$. In simulation and the following chapters q_{bead} is set to unity. Charges do not interact with one another.

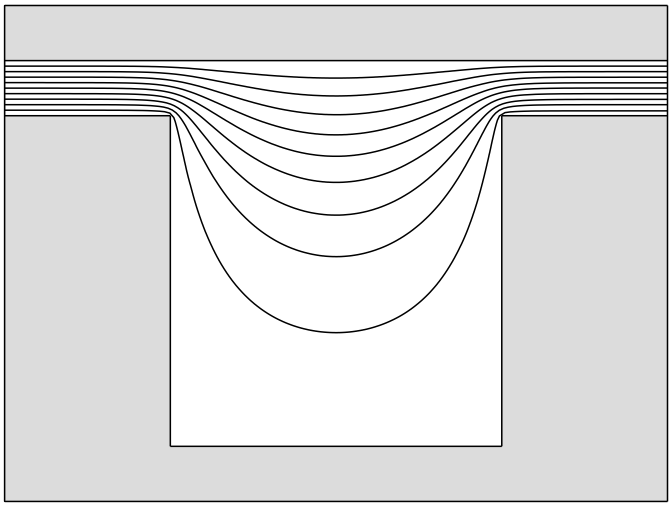

Figure 2.8: Electric field lines inside a sample device

Originally, the simulation program had been set up with a homogenous electric field. Trajectories obtained from simulation revealed unrealistic long durations of stay [86], which had not been reported before. In order to achieve results which were comparable to experiment, the walls of the device had been tilted. Finding the correct tilt angle remained very difficult and results were unfit for a detailed analysis.

However, a closer inspection reveals that the assumption of a homogenous electric field cannot be justified. The device contains solvent molecules and thus mobile ions inside, and consists of an insulator outside. The mobile charges will screen any local space charge (Eq. 2.32) and also screen any electric field perpendicular to the wall (Eq. 2.33). This leads to

$$\Delta\Phi = 0 \text{ inside the device} \tag{2.32}$$

$$\vec{n} \cdot \nabla\Phi = 0 \text{ at the walls} \tag{2.33}$$

Imposing a constant potential difference to a single device unit at the inlets now leads to the electric potential Φ. Then the electric field can be computed by taking the derivative of the potential Φ:

$$\vec{f}_{\mathrm{el}} \equiv \vec{E} = -\nabla\Phi \tag{2.34}$$

Note the electric field is not homogeneous. The electric field of a sample device can be found in fig. 2.8. Any given electric field refers to a potential difference applied to a device and is thus some kind of average electric field.

2.9.1 Pulsed electric Field

As will be shown in chapter 4, there are combinations of electric field and device geometry, which are not favorable for separation. However, these exhibit two distinct migration states. These are populated in different ratios depending on the electric field and the chain length. To exploit these two migration states, a pulsed electric field is applied to the device. A drawing of the pulsed electric field is given in figure 2.9. For a full

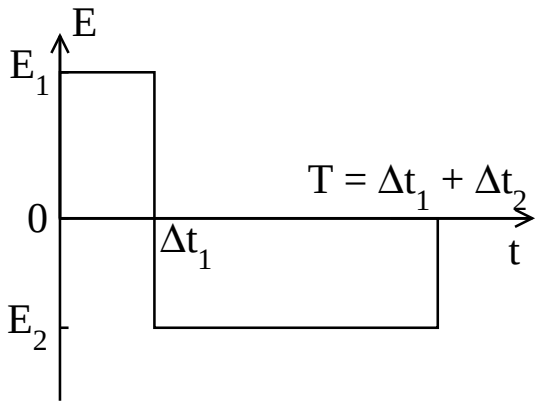

Figure 2.9: Schematic drawing of the pulsed electric field

description, only for parameters are necessary:

- The durations of the forward and backward pulse Δt_1 and Δt_2
- The electric fields applied during the two pulses E_1 and E_2

2.10 Neglected Interactions

Some important interactions are neglected by the simulation model. However, these should not affect the simulation data seriously, as will be discussed below.

2.10.1 Electrostatic Interaction

Each base pair of DNA is charged with $2e^-$. Thus base pairs repel each other. However, electrostatic interactions are screened with a Debye length of roughly 2nm, which is also roughly the diameter of the DNA strand. Taking into consideration the persistence length of about 45nm [10, 11], electrostatic interactions along the DNA strand are obviously neglectable.

2.10.2 Electroosmotic Flow

Another important effect of a charged object in solution is called electroosmosis. It occurs when charged objects in solution are subject to an external electric field. Charged objects

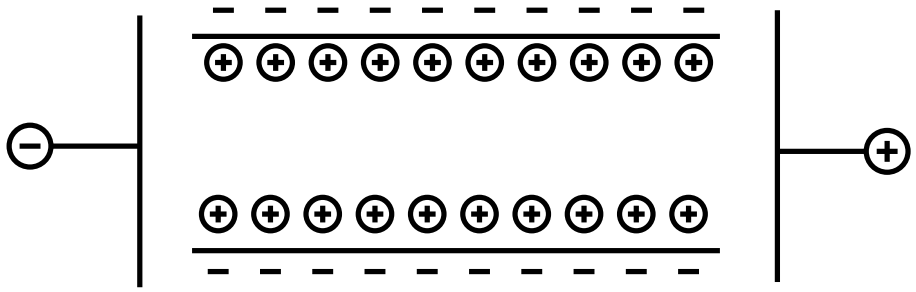

Figure 2.10: Schematic drawing of electroosmotic flow: the surface is negatively charged. Thus positive counterions are located near the walls. Application of an electric field leads to a net flow.

in solution are always surrounded by counter ions, which are mobile. Thus application of an electric field leads to a flow of the counter ions. My model does not consider electroosmotic flow, or flow in general.

The experimental device used in chapter 4 is made of PDMS, which exhibits silanol groups on its surface which dissociate under the experimental conditions. Thus the experiments were carried out in microchannels with negatively charged surfaces. This implies the generation of cathodic electroosmotic flow (EOF), which is shown in figure 2.10. The resulting mobility of DNA molecules is a sum of the electroosmotic and electrophoretic mobilities. The DNA molecules migrate to the anode, indicating that electrophoresis overcomes electroosmosis. However, recent experimental studies [99, 100] show that under appropriate conditions, the EOF outside the Debye layer at the walls of the device is directly proportional to the electric field. These are [100]:

- Low Reynolds number Re: Using the viscosity of water, an electroosmotic mobility $\mu_{\text{eof}} = (2.9 \pm 0.6) \cdot 10^{-4} \text{cm}^2\text{Vs}$ [63], an electric field of 100V/cm, and a characteristic length of 5μm, the Reynolds number turns out to be around 10^{-3}.

- Low product of Reynolds and Strouhal number $ReSt$: The Strouhal number describes the relaxation of the flow into the steady state. The EOF relaxation time is typically around 100μsec [45, 26], thus much shorter than experiments and neglectable.

- Zero applied pressure difference between all inlets and outlets.

- Uniform fluid properties.

- Radii of curvature small compared to the Debye layer thickness: The PDMS production process leads to slight deformations of the device, rounding off all corners (see [63], Fig. 1a).

- Insulating walls: PDMS is an insulator and this assumption is also used to compute the electric field.

Now, let α denote the proportionality constant between the electric field and the electroosmotic flow: $\vec{v}_{\text{eof}}(\vec{r}) = \alpha\vec{E}(\vec{r})$, with α independent of $\vec{r}$. Furthermore, let the force $\vec{f}_{\text{el}}$ denote the electric force and $\vec{f}_{\text{o}}$ denote all other forces (Lennard-Jones, spring, friction, walls) acting on a monomer. Hence the equation of motion for the velocity of the monomer i (Eq. 2.8) becomes:

$$\begin{aligned} m\dot{\vec{v}}_i &= \vec{f}_{\text{o},i} + \vec{f}_{\text{el},i} - \zeta(\vec{v}_i + \vec{v}_{\text{eof}}(\vec{r})) + \eta_i \\ &= \vec{f}_{\text{o},i} - (q + \zeta\alpha) \cdot \nabla\Phi(\vec{r}) + \eta_i, \end{aligned} \tag{2.35}$$

where equation. 2.34 is used. Note that this equation is formally equivalent to the original equation of motion (Eq. 2.8) with a rescaled charge q. In the overdamped case, the equation of motion (Eq. 2.15) becomes

$$\zeta\dot{\vec{r}}_i = \vec{f}_{\text{o},i} - (q + \zeta\alpha) \cdot \nabla\Phi(\vec{r}) + \eta_i, \tag{2.36}$$

which is again formally equivalent to the original equation of motion (Eq. 2.15). This implies that for a proper adaptation, the total mobility μ has to be considered. It is the sum of free flow mobility μ_0 and electroosmotic mobility μ_{eof}. Note that this similitude is not necessarily valid in the case of a time-dependent or high electric field.

2.10.3 Hydrodynamics

Hydrodynamic effects are not taken into account. This approximation must be questioned. On the one hand, DNA is always surrounded by counter ions, which are dragged into the opposite direction of the DNA. Thus the DNA molecule experiences not only hydrodynamic drag, but also an extra friction from the solvent molecules. In free solution, these two effects cancel each other [6]. This "hydrodynamic screening" accounts for the free-draining property of DNA. However, the total cancellation fails if the DNA molecule is blocked by a geometric barrier [101]. In that case, the counter ions will not be immobilized, since counterions are still free to move. Furthermore, hydrodynamic interactions affect the diffusion constant D. In our model, it scales with the chain length N like a Rouse chain ($D \propto 1/N$). Including hydrodynamic interactions, one would expect Zimm scaling ($D \propto 1/R_g \propto 1/N^{\nu}$). In experiments, the diffusion constant of DNA is found to scale as $D \propto 1/N^{0.672}$ [102].

3 Simulation of Migration in Entropic Trap Arrays

Parts of the results presented in this chapter have been published before by M. Streek, F. Schmid, T. T. Duong and A. Ros in ref. [58]. The analytical calculations presented in chapter 3.4.5, together with the necessary data evaluations, have been performed by Friederike Schmid.

3.1 Introduction

A successful separation by length has been demonstrated by Han and Craighead in a device called "entropic trap" [53, 54, 55, 56]. It is a periodic sequence of deep and shallow regions which have been etched into a silicon surface and is covered by a pyrex coverslip. The depth of the deep regions is around a few μm and thus allow for a coiled conformation. The shallow regions are only 90nm in depth, and force the polymer into an uncoiled state. A schematic drawing of the device is given in figure 3.1. To pass

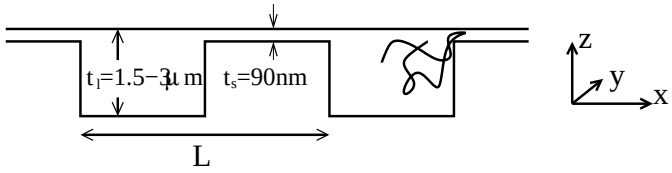

Figure 3.1: Schematic drawing of the device presented by Han and Craighead [55]. The width of the channel is much larger ($\approx 30\mu$m).

the shallow region, the polymers have to uncoil and thus overcome an entropic energy barrier ΔF_{C} (hence the name, "entropic trap"). Thus longer chains are expected to migrate *slower* than short ones, as the energy barrier is higher.

Experimentally, it is observed that long chains migrate *faster* than short ones. This implies that the simple picture given above cannot be applied in this way.

The energy barrier ΔF_{C} has to be overcome and the escape process is initiated by a thermal stretching (length x) of the polymer into the shallow constriction. On the one hand, penetration of the shallow region costs entropic free energy proportional to x [103]. This assumption remains valid in the "blob" picture as well [104]. On the other hand, there is an energy gain due to the electric field which is proportional to Ex^2, where E is the electric field [54]. Hence, there exists a critical length $x_{\mathrm{c}} \propto 1/E$, below which the entropic penalty dominates and the chain is driven back. For $x > x_{\mathrm{c}}$, the energy gain dominates and the chain is driven through the constriction. The energy barrier ΔF_{C} of

the escape process is solely described by the free energy at x_c, which depends only on the electric field E. The rate of escape attempts, $1/\tau_0$, increases with the chain size, since longer chains have a greater contact area with the constriction. In this simple model, the mean trapping time is given by [54]

$$\tau = \tau_0 \exp\left(\frac{\Delta F_C}{k_B T}\right). \tag{3.1}$$

This implies long chains migrate faster.

These results have motivated recent computer simulations. Tessier *et al.* have investigated the migration in entropic traps using the bond fluctuation model [57]. This model describes the polymer as a sequence of lattice sites which are either free or occupied. The results confirm the trapping picture by Han *et al.* [53, 54, 55, 56]. They even present evidence that the penetration depth x of the chain into the constriction can be used as a "reaction coordinate" for escaping, with a critical value $x_c \propto 1/E$. However, the simulation data reveal unexpected strong trapping of the polymer chains. Compared to the persistence length, the width of the constriction is almost twice as large as that used in experiment. This effect might be related to the lattice model used. The width of the constriction is just 10 lattice sites, and each monomer occupies a cube with 8 lattice sites. This leads to a persistence length of 2.8 lattice sites. Moreover, Monte Carlo (MC) is not appropriate to simulate time-dependent phenomena, or driven systems. MC simulations have been very successful for equilibrium systems, extensions to near-equilibrium systems are available. In this system, each monomer is pulled by the electric field. It is not clear how realistic MC moves describe directed motion, especially at high fields or outside the deep regions, in which the chains may reach equilibrium.

Another interesting work has been presented recently by Chen *et al.* [59]. They investigate the free energy land scape of a single escape process by an off-lattice bead-spring model with Monte Carlo methods. The initial configuration is that of a fully relaxed chain in the absence of an electric field. With this starting point, the energy barrier ΔF_C turns out to depend on the chain length for short chains, and levels off for longer chains. The data do not seem to support the relation presented by Han and Craighead, $\Delta F_C \propto 1/E$. The same limitations described for Monte Carlo simulations above also apply here. Furthermore, the relaxation time of long chains may easily exceed the migration time of a single device period. Thus it is not clear if back-to-back escape processes are uncorrelated.

3.2 Simulation Setup

For entropic traps, the width of the shallow regions is typically around 90nm [55]. This is comparable to the persistence length of the DNA molecule, which is around 45nm [10, 11]. The height of the shallow region is set to 7σ, which leads to an effective height of 5σ if the repulsive layer is subtracted. The length of the shallow region is set to 20σ, and the depth of the deep region is $(80 + 7)\sigma = 87\sigma$. The total length of the device is 100σ. In the lateral direction, the device is infinite. A snapshot of $N = 1000$ monomers is given

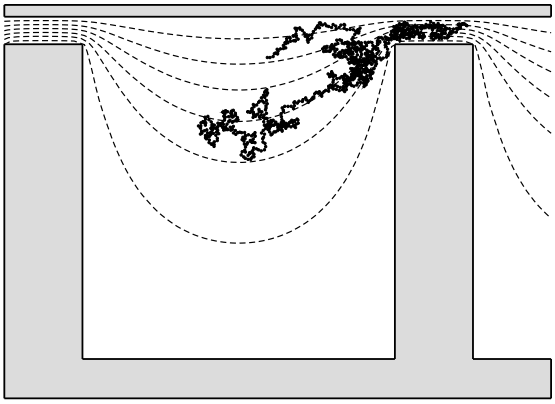

Figure 3.2: Snapshot of $N = 1000$ monomers in an entropic trap device. The electric field lines are shown as dashed lines.

in figure 3.2. This device has successfully been simulated before with a homogeneous electric field [86].

Simulations were performed at electric fields ranging from $0.0025 \ldots 0.04 E_0$. Run lengths varied from $4 \cdot 10^8 \ldots 2 \cdot 10^9$ with inertia dynamics, thus from 4 to 20 million t_0. The investigated chain lengths are $N = 10$, 20, 50, 100, 200, 500 and 1000 monomers.

3.3 Correspondence to Experimental Data

As the shallow regions exhibit a width comparable to the persistence length l_p, it is necessary to perform the length comparison with respect to this length scale. In this case, 45nm correspond to 1.6σ, or $\sigma \equiv 30$nm. The effective width of the shallow region is 5σ or $3l_\mathrm{p}$, which corresponds to 150nm. In experiment, the width of the constriction is 90nm, which is only twice the persistence length l_p, but still comparable. The depth of the deep region is $t_\mathrm{l} = 1 - 3\mu$m in experiment. In simulation, the depth is $t_\mathrm{l} = 85\sigma$, which corresponds to 2.6μm and compares well to the experimental value.

The persistence length is also used to determine the number of base pairs per bead. In this case, we have 1.9 beads per persistence length. A DNA molecule contains approximately 150 base pairs per persistence length. This leads to 1bead $\approx$ 80bp. Thus the simulation covers the range of $0.8 \ldots 80$kbp. Han and Craighead [55] investigated T2- and T7- DNA, which have 164 and 37.9 kilo base pairs, respectively. Thus the investigated chain lengths are comparable to each other.

The time t_0 is calculated from the diffusion constant D. For Rouse chains of length N, D is given by

$$D = \frac{k_\mathrm{B}T}{N\zeta} = \frac{\sigma^2}{Nt_0}. \tag{3.2}$$

Experimentally, Stellwagen *et. al* have reported the relation [102]

$$D = 7.73 \times 10^{-6} (\text{number of base pairs})^{-0.672} \mathrm{cm}^2 \mathrm{s}^{-1} \tag{3.3}$$

Choosing $N = 500$ beads (40kbp) as a reference, one obtains $t_0 \equiv 2.9 \times 10^{-6}$s. Simulation times cover the range from $4 \cdot 10^6 \ldots 2 \cdot 10^7$, thus covering experimental times from 10 to 60 seconds.

The mobility μ_0 can be found by matching the mobility of free chains. Unfortunately, Han and Craighead do not provide explicit measurements. They only present a maximum apparent mobility $\mu_{\max} \approx 0.13 \cdot 10^{-8}\mathrm{m}^2/\mathrm{Vs}$ [54]. Using the approximation that the electric field consists of two distinct field strengths, it is possible to estimate the free-flow mobility μ_0 [58]. Applying the formula presented leads to $\mu_0 = 0.55 \cdot 10^{-8}\mathrm{m}^2/\mathrm{Vs}$. This value is untypically low [102]. Furthermore, the free-flow mobility μ_0 is strongly dependent on the buffer [105]. Using the free-flow mobility estimate given above, one finds $E_0 \sim 2{\cdot}10^4\mathrm{V/cm}$. In experiment, electric field values ranging from 20 up to 80V/cm have been investigated. Assuming a free-flow mobility as reported by Stellwagen *et al.*, the electric fields correspond to $\sim 0.006 - 0.03E_0$ in simulation. Note this is just a rough order-of-magnitude estimate.

The model parameter that has yet to be determined is the mass m of a single monomer. Note this parameter does not influence the static properties of the polymer, like chain flexibility or radius of gyration. The mass of the monomers does, however, influence the vibrational modes of the polymer. One important property is the electrophoretic relaxation time τ_e. It determines the time scale, on which the initial velocity of a molecule in an electric field decays when it is suddenly switched off. In experiment, this time is around 10^{-9} to 10^{-12}s [42]. In simulation, the appropriate mass would be represented by $m \sim 10^{-3} - 10^{-6}\zeta t_0$. This implies that $\langle v^2 \rangle = 2k_BT/m$ becomes large and thus requires a small time step. To keep the simulation efficient, an unrealistic high mass of $m = 1\zeta t_0$ is used. However, both in simulation and in experiment, the properties of interest (like traveling times) take place on time scales much larger than τ_e, and do not depend on short-time details of the dynamics.

3.4 Simulation Results and Discussion

3.4.1 Trajectories from Simulation

Sample trajectories of $N = 10$, $N = 100$, and $N = 1000$ monomers are given in figure 3.3. The dashed horizontal lines show the onset of the shallow regions, and the dashed lines indicate the leading and the most backwards monomers. For $N = 10$, these cannot be distinguished from one another, even in the case of $N = 100$ this is hardly possible. However, these chains are quite small compared to the device dimensions.

For short chains ($N = 10$), the movement is dominated by diffusion. Occasionally, the chain diffuses even backwards by a whole device geometry (data not shown). The overall trajectory is quite irregular and the chain explores the whole device by diffusion. For chains of medium length ($N = 100$), the trajectory is still affected by diffusion, but it is much less pronounced. However, the chain may still get trapped for a considerable amount of time. In the case of long chains ($N = 1000$), the movement of the center of mass is almost smooth. Nevertheless, the movement of the single monomers is still affected by the device geometry, which is easy to see in figure 3.3c. A comparison with

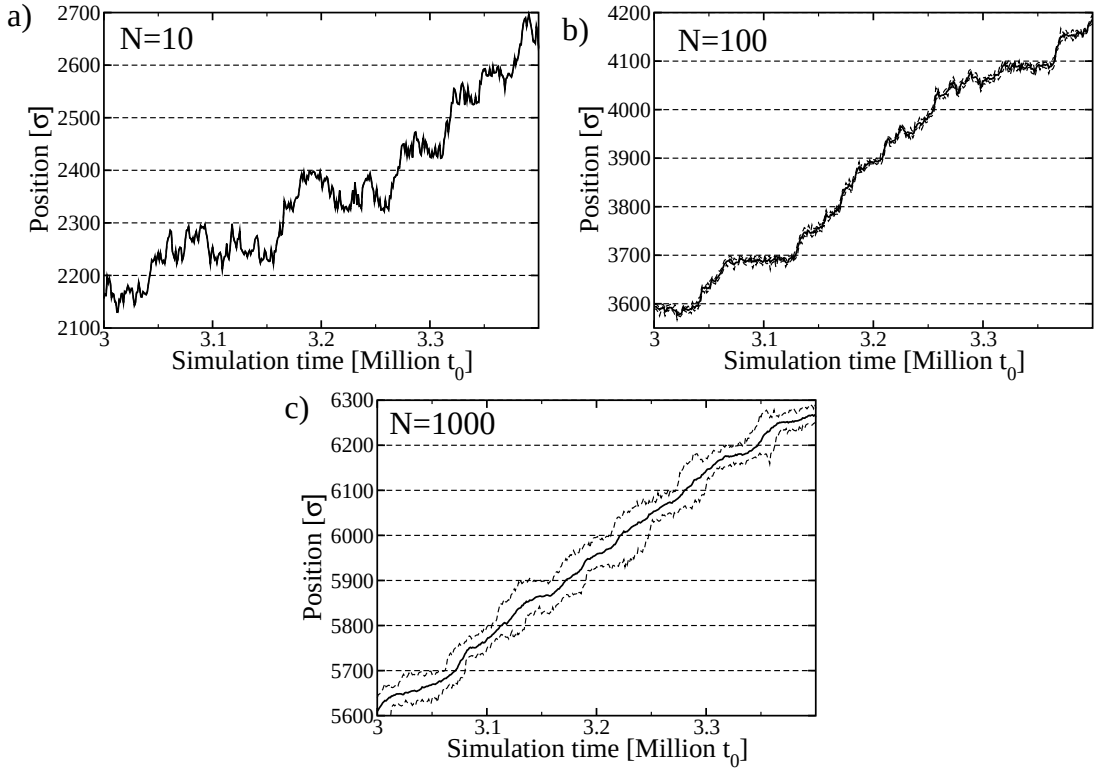

Figure 3.3: Trajectory of $N = 10$ (a), $N = 100$ (b), and $N = 1000$ (c) monomers at $E = 0.005E_0$ in an entropic trap device. The solid line shows the movement of the center of mass. The x-position of the leading and trailing monomers are indicated by dashed lines. The dashed horizontal lines represent the beginning of the shallow regions.

the simulation data obtained with homogeneous electric fields [86] shows that the strong trapping observed in devices with untilted walls has vanished.

Comparing these trajectories with the data presented by Tessier *et al.* [57], one sees that the trapping is comparable only in the case of $N = 100$. As will be shown below, an efficient separation requires stronger electric fields.

3.4.2 Relaxation of the Chains during Migration

Assuming a homogeneous electric field of $E = 0.01E_0$, a free chain is expected to migrate through the device in $L/E_0 = 10^4 t_0$. Using equation 2.29, the associated chain length is $N_c \approx 270$ monomers. Thus chains longer than N_c are unable to relax during the migration through the device. This implies the picture presented by Chen *et al.* [59] of independent escape processes is no longer valid. The normalized expectation value of the radius of gyration, split into its components, inside the device is shown in figure 3.4.

All chains are stretched considerably in x-direction when passing the shallow region.

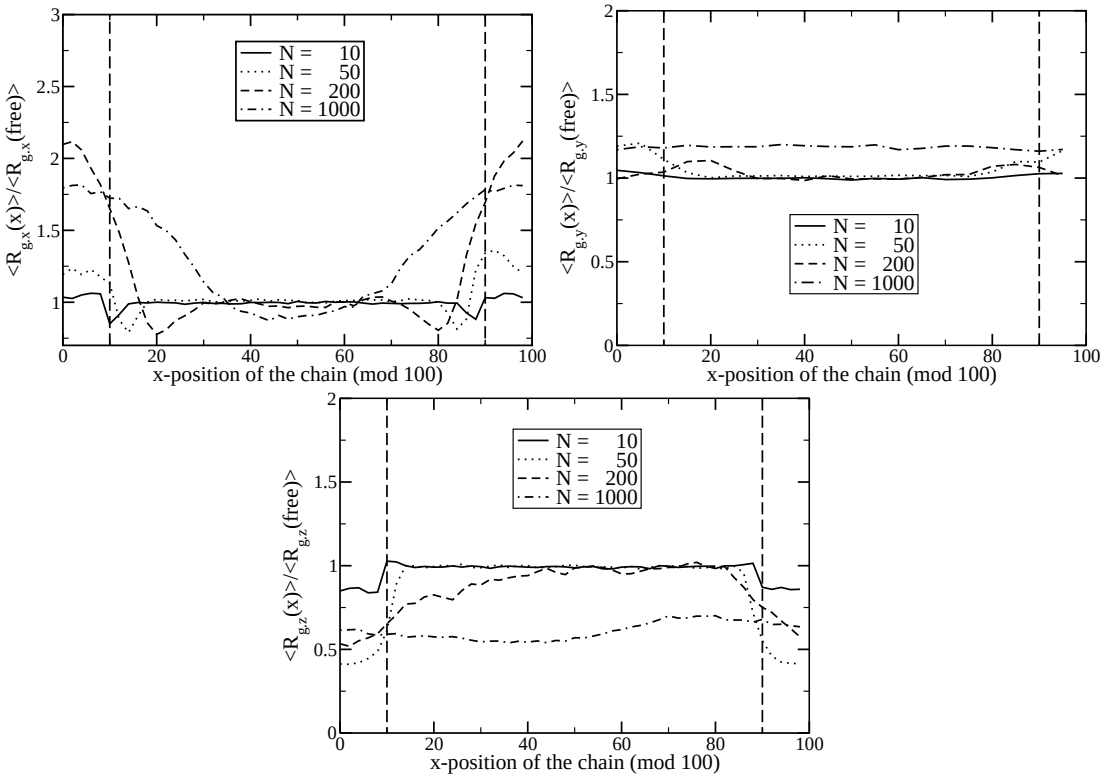

Figure 3.4: Normalized components of R_g inside the Device. Short chains are able to relax when passing the deep region, whereas long chains are unable to do so. The limits of the deep region in the middle are shown by the vertical long dashed lines.

In the case of $N = 10$ monomers, the stretching is very small, because the shallow region itself is longer than the contour length of the chain.

In the y-direction, the chains are mostly unaffected by the device. This is hardly surprising, as the device is infinite in that direction. However, when passing the shallow region, the chains do get stretched because of the stress exerted onto the chain by the device. In the case of $N = 1000$ monomers, the chain is stretched permanently, and does not recover its size in free diffusion.

When passing the shallow region, all chains are limited in the z-direction. Therefore, all chains are squeezed when passing this region. However, except in the case of $N = 1000$ monomers, the chains are able to regain their free size, only the chain with $N = 1000$ monomers is unable to adapt to the unperturbed size.

To summarize, the simulation data prove the ideas presented above. Whereas short chains are able to relax in the deep regions, long chains do not regain their unperturbed properties during migration, even if the device would allow for a coiled conformation [86].

3.4.3 Mobilities from Simulation

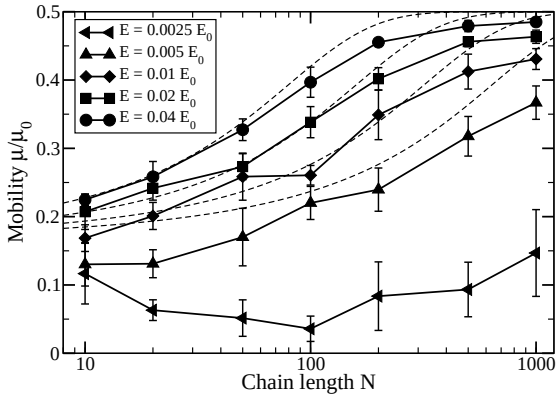

Figure 3.5: Mobility μ in units of the free-flow mobility μ_0 as a function of the chain length N, for various electric fields E. The dashed lines show the approximation given in equation 3.10, for $E = 0.04E_0$, $0.02E_0$, $0.01E_0$, and $0.005E_0$ (top to bottom).

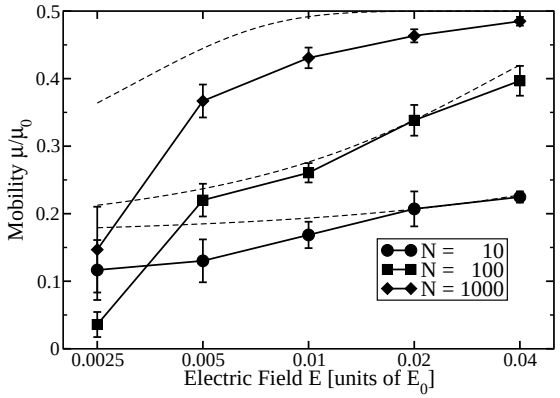

Figure 3.6: Mobility μ in units of the free-flow mobility μ_0 as a function of the electric field E, for various chain lengths N. The dashed lines show the approximation given in equation 3.10, for $N = 1000$, 100, and 10 (top to bottom).

From the simulation data described above, it is easy to compute the mobility μ of a chain with N monomers in the device for a given electric field E:

$$\mu = \langle v \rangle / E. \tag{3.4}$$

Mobilities obtained by this method are given in figure 3.5 as a function of the chain length N, for various electric fields E. In the case of the lowest field $E = 0.0025E_0$, the mobility depends only slightly on the chain length and decreases with chain length for

small N. This can be explained by the fact that short chains explore the whole device. Therefore, in the limit $E \to 0$, the escape rate depends only on the diffusion constant D, which decreases for increasing chain length. For stronger electric fields, the mobility increases with the chain length and apparently levels off at $\mu_{\max} \approx 0.5\mu_0$.

The dependence of the mobility μ on the electric field E is shown in figure 3.6 for various chain lengths N. The mobility μ increases as a function of the electric field E for all chain lengths. However, for $N = 10$ monomers the dependence is only slight. This can be explained by the fact that short chains underly a strong diffusion, which actually dominates the escape process.

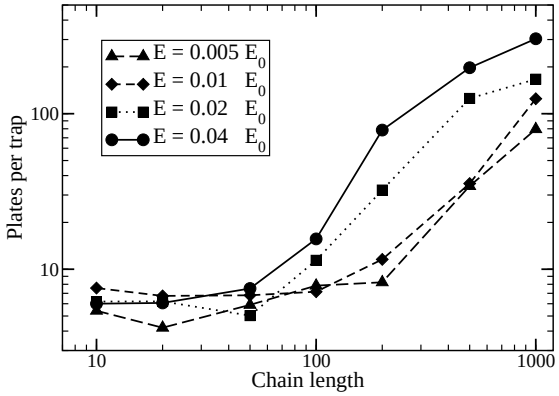

Figure 3.7: Theoretical plate number per trap as a function of the chain length N for different electric fields

The quality of molecular separation can be determined in terms of the theoretical plate number. It is defined as

$$N_{\text{plate}} = 16(t_{\text{R}}/t_{\text{W}})^2, \tag{3.5}$$

with the total retention time t_{R}, *i. e.*, the total time spent in the system, and t_{W} is the width of the peak at the baseline. The results of this analysis are given in figure 3.7. In the interesting regime, typical plate numbers are around $10 - 100$. Assuming $\sim 10^5$ traps per meter, this corresponds to $10^6 - 10^7$ plates per meter. This is quite good and in agreement with the data presented by Han *et al.* [55].

3.4.4 Detailed Analysis of the Migration

To achieve a deeper understanding of the migration, the retention times of the different chain lengths in the device were investigated. These were defined as the time difference $t_{n+1} - t_n$ of the times t_n when the leading monomer first entered the deep region. A distribution of the data obtained for $E = 0.01E_0$ is given in figure 3.8. After an initial "dead" time of about $\sim (1 - 1.5) \cdot 10^4 t_0$, the chains start leaving the trap. This time can easily be related to length of the device, as already described in chapter 3.4.2. After that, the histogram rises rapidly and reaches a maximum at $t_{\max} \approx 2 \cdot 10^4 t_0$. A

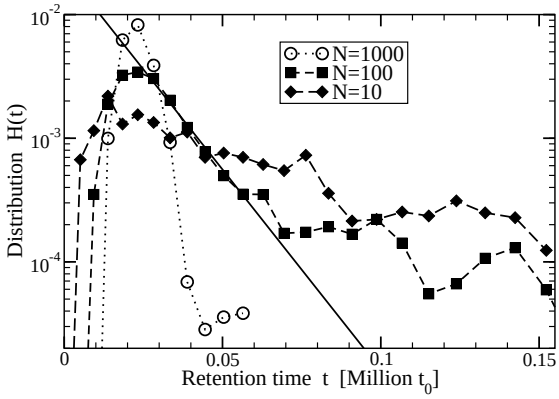

Figure 3.8: Distribution of retention times in one trap for different chain lengths at $E = 0.01E_0$. The solid line is a fit to the initial exponential decay at chain length $N = 100$

comparison of all histograms yields that the maximum of the distribution $t_{\max}$ is almost independent on the chain length N. However, it is proportional to the inverse of the electric field $1/E$, which can be directly related to the migration time. Apparently, the product $L/t_{\max}E = \mu_{\max} \approx 0.5\mu_0$ is fulfilled for sufficient electric fields, which can be seen in figure 3.9. The decay of the distribution after the maximum is not independent of the electric field.

Beyond the maximum, the histogram decays rapidly for $N = 1000$. In the case of $N = 10$, the decay is much slower and follows an exponential law. This is characteristic for the existence of a single escape rate $1/\tau$, as described by Han *et al.* [53, 54, 55, 56]. In the case $N = 100$, this simple picture is no longer applicable. After the maximum, the data indicates an initial exponential decay, which is shown in figure 3.8 with a solid line. For long times, the data does not match the fit any more. Apparently, there is another trapping mechanism, which has its own timescale, τ_{slow}, which differs from the first, τ_{fast}.

These two mechanisms can already be guessed from the trajectory presented in figure 3.3b. On the one hand, the chain passes many traps smoothly without getting trapped for a long time. Occasionally, however, it gets stuck for a reasonable amount of time. In that case, it gets stuck at the border of the constriction.

As the trapping mechanism presented by Han *et al.* exhibits only one time scale τ, it cannot explain this behavior alone. Thus a detailed look at possible trapping mechanisms is necessary and will be given in the following sections.

3.4.5 Trapping in the deep Region of the Device

As seen in chapter 3.4.4, the trapping mechanism proposed by Han *et al.* [53, 54, 55, 56] alone cannot explain the results obtained. Here, we present a second trapping mechanism, which also favors fast migration of long chains. A schematic drawing of the two

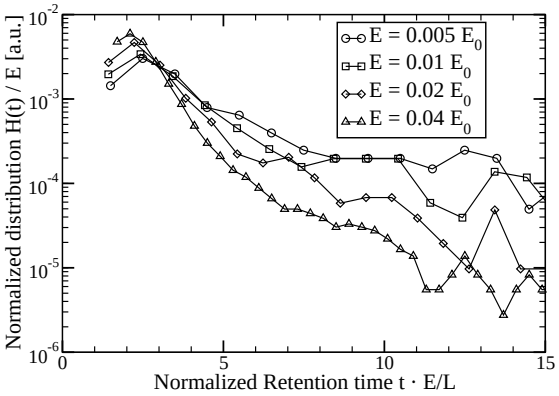

Figure 3.9: Distribution of retention times in one trap for different electric fields at chain length $N = 100$. The data has been normalized. All distributions exhibit a maximum at $t_{\max} \approx 2L/E$.

trapping mechanisms is shown in figure 3.10:

(a) This is the mechanism presented by Han *et al.*. This mechanism is related to the time τ_{fast}.

(b) Chains may also get trapped in the field-free region in the lower part of the device. Here, the chain experiences almost no force, and it may leave this region only by diffusion. Furthermore, it is able to form a coil. This escape process is related to the time τ_{slow}. Leaving this region implies stress on the coil due to inhomogeneous electric field, or moving against the electric field. Hence there is an entropic or energetic penalty for leaving this region. To enter this region, the chains have to be sidetracked from the field lines considerably. Thus short chains exhibit a greater probability to get stuck.

We will now investigate the data in more detail and check if the data supports this picture. In the following analysis, only data for $N \geq 20$ and $E \geq 0.005E_0$ were analyzed.

Indeed, two time scales were found in most of the analyzed systems. It turned out that the fast time scale, τ_{fast}, could be determined quite easy by fitting the initial decay distribution $H(t)$ with an exponential function ($A\exp(-t/\tau_{\text{fast}})$). Unfortunately, the slow time scale, τ_{slow}, is much harder to obtain. The reason is simply the poor statistics of the late time tails of the distribution $H(t)$. Therefore, this time scale was obtained in two different ways: First, an exponential function was fitted to the long tail. The obtained value was used as a rough estimate. Another way to obtain τ_{slow}) is to assume that $H(t) \propto \mathrm{e}^{-t/\tau}$. Then, a cut off for the decay time was introduced, t_{cut}, and only data for $t \geq t_{\text{cut}}$ was taken into consideration. The slow decay time τ_{slow} can now be

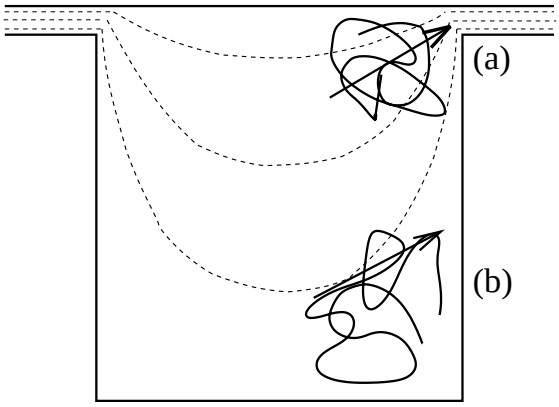

Figure 3.10: Trapping mechanisms in the device. Chains may get stuck at the entrance of the constriction (a). Furthermore, chains may get stuck in the field-free region in the lower parts of the device (b). These chains experience a net force towards the wall.

determined by

$$\begin{aligned}\frac{\int_{t_{\mathrm{cut}}}^{\infty} \mathrm{d}t(t-t_{\mathrm{cut}})H(t)}{\int_{t_{\mathrm{cut}}}^{\infty} \mathrm{d}t H(t)} &= \frac{\int_{t_{\mathrm{cut}}}^{\infty} \mathrm{d}t\, tH(t)}{\int_{t_{\mathrm{cut}}}^{\infty} \mathrm{d}t\, H(t)} - t_{\mathrm{cut}} \\ &= \frac{t_{\mathrm{cut}}\tau_{\mathrm{slow}} \exp(-t_{\mathrm{cut}}/\tau_{\mathrm{slow}}) + \tau_{\mathrm{slow}}^2 \exp(-t_{\mathrm{cut}}/\tau_{\mathrm{slow}})}{\tau_{\mathrm{slow}} \exp(-t_{\mathrm{cut}}/\tau_{\mathrm{slow}})} - t_{\mathrm{cut}} \\ &= \tau_{\mathrm{slow}},\end{aligned} \tag{3.6}$$

which is independent of t_{cut}. For analysis, we set $t_{\mathrm{cut}} = 500 t_0 E_0/E$. This result was tested against the previously obtained estimate. If it deviated too strong, it was discarded.

If the suspicion presented above is correct, the fast time scale τ_{fast} is corresponds to the mechanism presented by Han *et al.*. In this case, the escape rate $1/\tau_{\mathrm{fast}}$ is proportional to the amount of polymer which in contact with the wall. As the entrance of the constrictions is basically one dimensional, the escape rate should be proportional to the radius of gyration, R_{g}. This leads to $\tau_{\mathrm{fast}} \propto N^{-3/5}$. The relation between τ_{fast} and E is more complex and not as easy to determine. According to Han *et al.*, the chains have to overcome a free energy barrier $\Delta F \propto 1/E$ when escaping. On the other hand, equation 3.1 does not determine τ_0. Thus the resulting E-dependence may be complex.

The results from this analysis are given in figure 3.11, together with a power law as indicated. The fit for $\tau_{\mathrm{fast}} \propto N^{-3/5}$ fits very well. With my data, we find that $\tau_{\mathrm{fast}} \propto E^{-1.55}$ by empiric evaluation.

The slow time scale τ_{slow} decays with the chain length N, but the dependence is weak. Furthermore, the quality of the data is poor, and no quantitative analysis is possible. If the picture presented above is correct, τ_{slow} is related to the escape from the low-field region of the trap. In this region, chains may form a coil and thus experience a weak

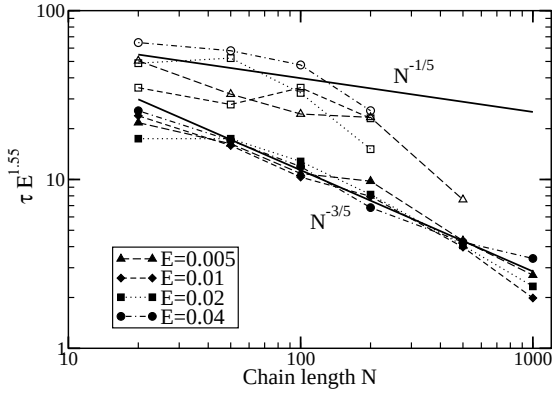

Figure 3.11: Characteristic time scales (rescaled) of the retention time distribution for different chain lengths, N, and different electric fields, E. Filled symbols correspond to the fast time scale, τ_{fast}, and open symbols refer to the slow time scale, τ_{slow}. The thick lines show power laws for comparison, as indicated.

electric field which results into a net force towards the wall (Fig. 3.10b). The escape process from this area implies either uncoiling due to the inhomogeneous electric field, or moving against the electric field. In both cases, there is an energy barrier (either entropic or electric), before the chain is pulled to the entrance of the constriction by the electric field. A simple Ansatz predicts that the escape probability is proportional to the diffusion constant, D, and diffusion by either one radius of gyration R_{g} or the total chain length N is necessary. This results in a chain length dependence of either $\tau_{\text{slow}} \propto N^{-1/5}$, or $\tau_{\text{slow}} \propto N^0$. The data obtained is consistent with these results (Fig. 3.11).

The chain length dependence of the slow time scale τ_{slow} is small. On the other hand, the probability of getting caught depends on the chain length, as the *relative* number of chains getting caught strongly depends on the chain length N. We use the assumption that the travel time from one trap to the next one is roughly

$$t_{\max} = L/E\mu_{\max}, \tag{3.7}$$

and that the chains have to diffuse a minimum distance z_0 into the deep region of the trap. After the time $t_{\max}$, the distribution of the center of mass in the z-direction is roughly Gaussian: $N(z) \propto \exp(-z^2/6Dt_{\max})$. Thus the probability of getting caught can be estimated:

$$P = \int_{z_0}^{\infty} \mathrm{d}z\; N(z) = \operatorname{erfc}(z_0/\sqrt{6Dt_{\max}}) = \operatorname{erfc}(\alpha\sqrt{NE}), \tag{3.8}$$

where $\operatorname{erfc}(y)$ is the complementary error function $\operatorname{erfc}(y) = 2/\sqrt{\pi}\int_y^{\infty} \mathrm{d}x\; \exp(-x^2)$.

Using the assumption that the two time scales, τ_{slow} and τ_{fast} are sufficiently apart from each other, it is possible to perform a simple analysis, as is shown below. In this case, all the chains in the fast mode have left the trap after a time t_{cut}, and no chains

in the slow mode have left the trap before t_{cut}. Thus only fast escape processes provide a noteworthy contribution to the decay. Then an approximation of P_0 is possible, and it is given by the fraction of chains that are left in the trap after t_{cut}. The results of

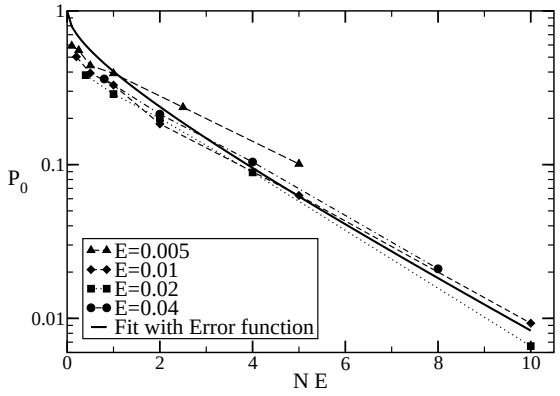

Figure 3.12: Probability $P_0(t)$ that chains are still in the deep region after $t_{\mathrm{cut}} = 350/E(t_0 E_0)$, shown as a function of NE, in units of E_0. The solid line shows a fit to equation 3.8, with $\alpha = 0.59$

this analysis are shown in figure 3.12, the solid line a fit of the data to equation. 3.8. The data collapse is good and the fit function agrees well with the data for $\alpha = 0.59$. Substituting equation 3.7 into equation 3.8, it is easy to find

$$z_0 = \alpha\sqrt{6DtNE} = \alpha\sqrt{\frac{6L}{\mu_{max}}}. \tag{3.9}$$

Inserting $\mu_{\max} = 0.5\mu_0$ and $L = 100\sigma$, one finds that $z_0 \sim 20\sigma$. Thus chains get caught in the deep region, if they diffuse more than 20σ from the main path into the deep region, which is defined by the electric field lines.

A detailed plot showing both trapping mechanisms is given in figure 3.13. In the insets, a side-view of the migration over the length indicated by the dashed lines is shown:

(a) In the fast escape process, the polymer trajectory does not deviate considerably from the main path, and the migration is rather smooth.

(b) During the slow escape process, the polymer indeed deviates from the main path, and gets trapped for a rather long time. In this case, it may explore the whole trap.

The considerations presented above establish a new trapping mechanism, which also produces chain length dependent behavior and favors long chains. To evaluate the importance of the new mechanism, let us consider a very simple model and compare the results with the simulation data. In this model, chains either travel smooth, or get sidetracked and caught in the field-free region of the trap. In the first case, all

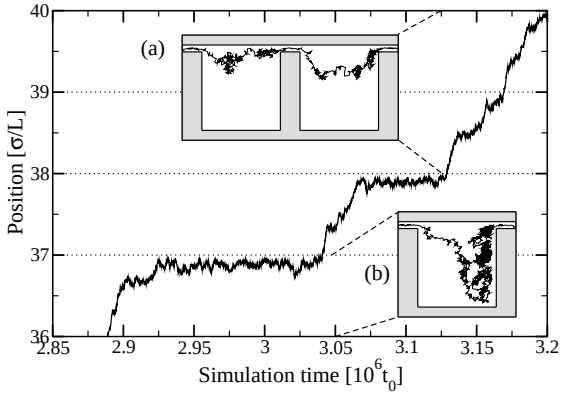

Figure 3.13: Detailed trajectory of $N = 100$ monomers at $E = 0.005E_0$. The entrances of the constrictions are marked by dotted lines. The insets show a side view of the movement inside the trap, as indicated by the dashed lines. Chains either move along the electric field lines (a), or get trapped in the field-free region at the bottom (b). In that case, the escape rate is very low.

chains need the minimum travel time $t_{\max} = L/E\mu_{\max}$ (Eq. 3.7). If chains get caught in the field-free region, they spend an additional time τ_{slow} in the deep region, thus the total time spent in the trap is $t_{\max} + \tau_{\text{slow}}$. We use the simplification that the product $E\tau_{\text{slow}}$ is independent of both chain length N, and electric field E and assume that it is given by $E\tau_{\text{slow}} = 400E_0t_0$. This is roughly the value obtained for $N \sim 100$, and $E = 0.01 - 0.02E_0$. The relative number of chains caught in the deep region is computed by equation 3.8, with α set to 0.59. In this model, the resulting mobility is given by

$$\frac{\mu}{\mu_0} = \left[\frac{\mu_0}{\mu_{\max}} + \frac{\tau_{\text{slow}}E}{L}\text{erfc}(\alpha\sqrt{NE})\right]^{-1}. \tag{3.10}$$

A comparison with real mobility data obtained by simulation is given in figures 3.5 and 3.6. The agreement is remarkably good, despite the simplicity of the model. The agreement is very good for high electric fields $E = 0.04E_0$. For low fields, the agreement is not as good, which can probably be related to the fact that at low fields, $E = 0.0025E_0$, the separation becomes inefficient.

3.5 Conclusions

To conclude this chapter, I have presented the first off-lattice Brownian dynamics simulation of DNA migration in an entropic trap array. Even with the simple model used, it is possible to reproduce the experimental behavior, *i. .e.*, that long chains migrate faster than short chains. This effect can be tracked down to two distinct trapping mechanisms. The first mechanism has been presented and discussed by Han *et al.* [53, 54, 55, 56] before. Chains are trapped at the entrance of the constriction, and need to overcome a

barrier in the entropic free energy. This effect leads to an escape rate proportional to the radius of gyration of the chain, and thus scales as $N^{3/5}$. However, this effect accounts only for part of the chain length dependence presented here. A detailed analysis of the migration data revealed a second time scale, and thus a second trapping mechanism: chains may get stuck in the deep region of the trap. In that region, there is almost no electric field, and chain may only escape by diffusion. Thus trapping times may become quite long, and in this case almost independent on the chain length. The chain length dependence of the second mechanism comes from the fact that the probability of getting caught is dependent on the diffusion constant, and decreases with chain length.

To our best knowledge, this mechanism has not been described in literature before. This mechanism becomes relevant, if the period L of the device becomes small. Han *et al.* have investigated device lengths between $4 - 40\mu$m, but they reported trapping only for the shortest geometry, $L = 4\mu$m. Thus, the geometry presented here is comparable to the device used by Han *et al.*

My data further proves that the picture of independent escape processes proposed by Chen *et al.* [59] cannot be applied here. In my simulations, I present true non-equilibrium systems. Long chains do not recover the ideal coil structure in the deep regions, even if the device dimensions suggest so. The simulation data suggests that the chains do have a memory of the previous escape process, as successive escape processes seem to be correlated, especially at high fields. Unfortunately, the data obtained does not allow for a thorough analysis, because the statistics is too poor.

4 Two-State Migration

Parts of the results presented in this chapter have been published by Martin Streek, Friederike Schmid, Thanh Tu Duong, Dario Anselmetti, and Alexandra Ros in reference [63], which at the time of writing has been accepted for publication by Physical Review E. An extended abstract can be found in reference [64].

This work was created in a collaboration of the Condensed Matter and the Experimental Biophysics and applied Nanosciences Groups at the University of Bielefeld, Germany.

4.1 Introduction

In this chapter, I will investigate a device presented by Duong *et al.* [62, 64], which also is capable of a successful separation. It is very similar to entropic traps [53, 54, 55, 56], as it also consists of a periodic sequence of wide and narrow regions. The main difference

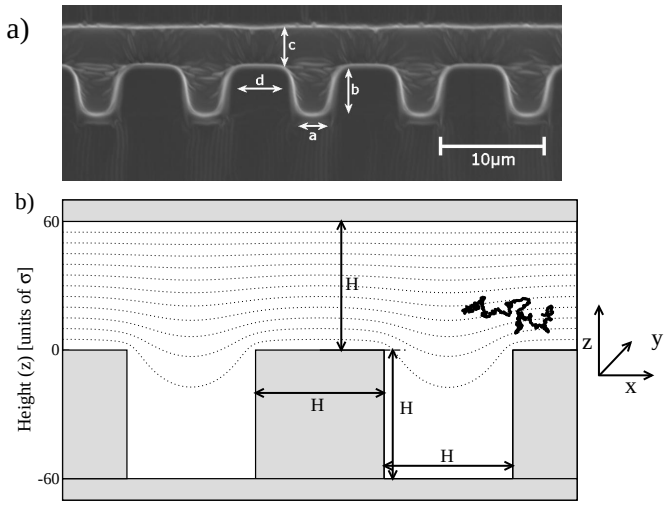

Figure 4.1: (a) SEM image of the experimental device [63], and (b) Schematic drawing of the investigated device. All size parameters of the device are equal. Thus only one parameter describes the geometry, the height of the thin region, H. The dotted lines represent the electric field lines, and the coordinate system used in the simulations is given on the right. Also shown is a simulation snapshot of $N = 320$ monomers.

compared to entropic traps is that width of the narrow regions is a few μm, which is much larger than the persistence length. Furthermore, only one parameter is necessary to

describe the device, the height H of the narrow region, as all device parameters are equal. An SEM image of the experimental device and a schematic drawing of the investigated device are shown in figure 4.1. Experimentally, the device has been integrated into a PDMS surface, and is covered by a glass plate. This device has the nice feature that it is possible to observe chain migration from the side and thus investigate chain movements in more detail than in the entropic traps presented by Han *et al.*, who were only able to look at chain migration from above. Details of the production process are given in reference [62, 63].

In this device, the trapping mechanism observed by Han *et al.* [53, 54, 55, 56] cannot be applied, because the narrow regions do allow for a coiled conformation of the polymer. However, the second mechanism, which is described in chapter 3.4.5, does not depend on extremely narrow channels, and can be applied here. Therefore, one would naively expect long chains to migrate *faster* than short ones.

Indeed, Duong *et al.* [62] have reported length dependant mobility in these structured microchannels. Unexpectedly, the migration order observed is inverse to the one observed by Han *et al.* In these microchannels, long chains migrate *slower* than short ones. In this chapter, I will investigate the reasons for this unforeseen behavior.

4.2 Simulation Setup

Duong *et al.* have investigated structures with channel heights $H = 1.5$, 3, and 5μm [61], and length dependent mobilities have been reported for $H = 1, 5\mu$m and $H = 3\mu$m [62]. In the lateral direction, all channels are limited to a thickness of $2.8\mu m$. During the production process, PDMS slightly changes its shape. Therefore, channel exhibit more than one length parameter, which can be seen in figure 4.1a. Details on the device dimensions in experiment are given in references [62, 63].

In the simulations, I use the idealization that all channel dimensions are equal and comparable to the radius of gyration of the investigated chains. I set the channel width of the narrow region to $H = 60\sigma$, which is also true for the limitations in the lateral direction. Thus the repulsive layer covering the walls (Ch. 3.3) becomes neglectable. The electric fields investigated cover the range from $E = 0.0005 - 1E_0$. The electric field lines are shown in figure 4.1b.

Simulations have been carried out covering times from $4 - 20 \cdot 10^8 \Delta_\mathrm{t}$, or $4 - 20 \cdot 10^6 t_0$ for inertia dynamics. For equilibration, $2 \cdot 10^7 \Delta_\mathrm{t}$ were used. Chain lengths investigated covered the range from $N = 10$ up $N = 2000$ monomers.

In some cases, simulations in the overdamped limit ($m \to 0$ or $\tau \to 0$) were carried out for comparison. In that case, simulation was carried out for $4{\cdot}10^8 \Delta_\mathrm{t}$, or $4{\cdot}10^4 t_0$. Starting configurations were taken from equilibrated systems with inertia dynamics. Due to the reduced time step, chain lengths investigated cover the smaller range from $N = 10$ up to $N = 100$ monomers. Electric fields investigated are $E = 0.04$, 0.08, 0.15, 0.25, 0.5, and $1E_0$.

4.3 Correspondence to experimental Data

The correspondence of the energies has already been described in chapter 2.4, and it yields $\in \equiv 300 k_B K \approx 0.026$eV.

In this case, the width of the narrow regions is comparable to the radius of gyration, which is also true for all other length parameters. Therefore, the persistence length is not a relevant length scale, because all device parameters are much larger. This leads to a comparison of the channel heights, H. Note that this implies that it is possible to match *all* three experimental devices to the *single* geometry of the simulation device, if the parameters are adjusted accordingly. In simulation, I used $H = 60\sigma$. In the case of 5μm structures, this leads to $1\sigma \equiv 83$nm.

Next, a comparison of the radii of gyration is performed. For comparison, λ-DNA (48 kbp, $R_g = 0.7\mu$m [106]) was used. Using $\sigma \equiv 83$nm from above, one finds that λ-DNA is represented by a chain with $R_g = 8.4\sigma$. Using the free-flow simulation data obtained earlier [86] yields that λ-DNA is represented by 140 beads, or 1 bead $\equiv$ 340 bp.

In the third stage, the time scale t_0 is adjusted to experimental time. Here, I compare the diffusion constants, D. Experimentally, Smith *et al.* [9] have reported $D_\lambda = (0.47 \pm 0.03)\mu\text{m}^2/\text{s}$. In simulation, D is given by $\frac{k_B T}{N\zeta}$. Matching the time it takes the polymer to diffuse along its own radius of gyration, I find that $1\text{sec} \equiv 9.5 \cdot 10^3 t_0$. Note that this time is the same as the conformational relaxation time (Ch. 2.7.1).

At last, the electric field is adjusted by comparing the mobilities of the chains. Note that it is necessary to use the absolute mobility to take electroosmotic flow into account (Ch. 2.10.2). In the simulation, the mobility μ_0 is simply the inverse of the friction, $\zeta = 1/\mu_0$. In experiments, Duong *et al.* [62] have found $\mu_0 = 1.84 \cdot 10^{-4}\text{cm}^2/\text{s}$. Comparing the time it takes the polymer to migrate a fixed distance to the mean quadratic drift induced by diffusion during that migration leads to the time correspondence $E_0 \equiv 430$V/cm.

These values, together with the values obtained for $H = 3$ and 1.5μm, are given in table 4.1. Adaptations given in chapter 4.4 refer to the 3μm channels. All other adaptations in this chapter refer to the 5μm channels, unless stated otherwise.

	$H = 1.5\mu$m	$H = 3\mu$m	$H = 5\mu$m
1 bead	48 bp	150 bp	340 bp
λ-DNA	1000 beads	330 beads	140 beads
T2-DNA	3500 beads	1200 beads	490 beads
1μm	40σ	20σ	12σ
1s	$7.5 \cdot 10^5 t_0$	$6.2 \cdot 10^4 t_0$	$9.5 \cdot 10^3 t_0$
1V/cm	$9.8 \cdot 10^{-5} E_0$	$5.9 \cdot 10^{-4} E_0$	$2.3 \cdot 10^{-3} E_0$
E_0	10kV/cm	1.7kV/cm	430V/cm

Table 4.1: Adaptation of the simulation units to various channel sizes. All values have been rounded to two digits.

4.4 Simulation at low electric Field

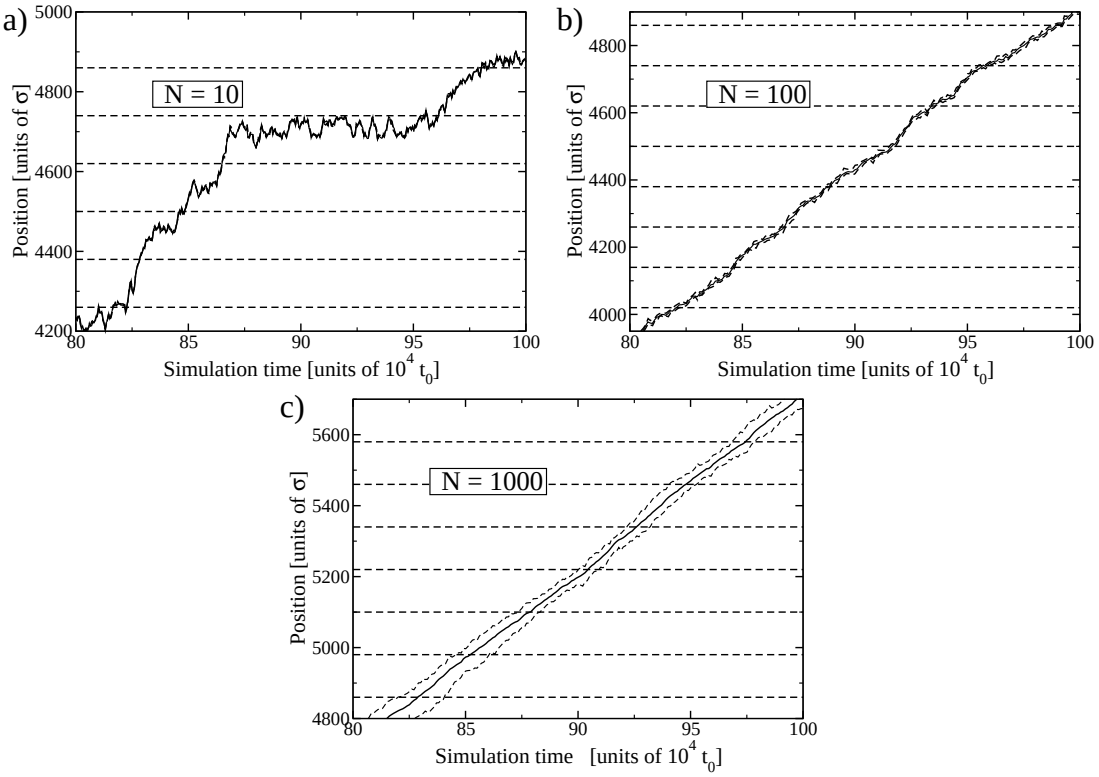

Figure 4.2: Trajectory of $N = 10$ (a), $N = 100$ (b), and $N = 1000$ (c) monomers at $E = 0.005E_0$ in the structured microchannel. The beginning of the narrow regions is marked by dashed, horizontal lines. The solid line show the center of mass, and the dashed lines show the leading and trailing monomers. For short chains (a, b), these cannot be distinguished from one another.

Duong *et al.* have presented successful separation in the 1.5 and 3μm channels [62]. Investigated electric fields covered the range from $10 - 100$V/cm. In simulation, I used electric fields of 0.0025, 0.005, 0.01, 0.02, and $0.04E_0$. This corresponds $4.3 - 68$V/cm in experiment, when the 3μm adaptation is used. As can be seen in table 4.1, this corresponds to rather low fields in larger structures. Chain lengths cover the range from $N = 10$ up to $N = 2000$ monomers, which corresponds to DNA strands with $1.5 - 290$kbp. The investigated DNA strands exhibit lengths of 48kbp in the case of λ- and 164kbp in the case of T2-DNA. Thus the simulation data covers the experimental ranges for both the chain length and the electric field.

Sample trajectories of the simulation are given in figure 4.2 for various chain lengths. In the case of $N = 10$ (a), the trajectory is dominated by diffusive motion, and is quite irregular on short time scales. Sometimes, the chain even moves back by one device

length. The trajectory looks very similar to that presented in figure 3.3a. In the case $N = 100$ (b), the trajectory is much smoother, but occasionally, trapping may still occur. For long chains, $N = 1000$ (c), the trajectory is very smooth, even the leading and trailing monomers do not indicate much interaction with the walls. This emphasizes the new trapping mechanism presented in chapter 3, because the mechanism presented by Han *et al.* does not apply here. The narrow regions do allow for a coiled conformation and do not force the polymer to uncoil. However, trapping in the deep region may still occur.

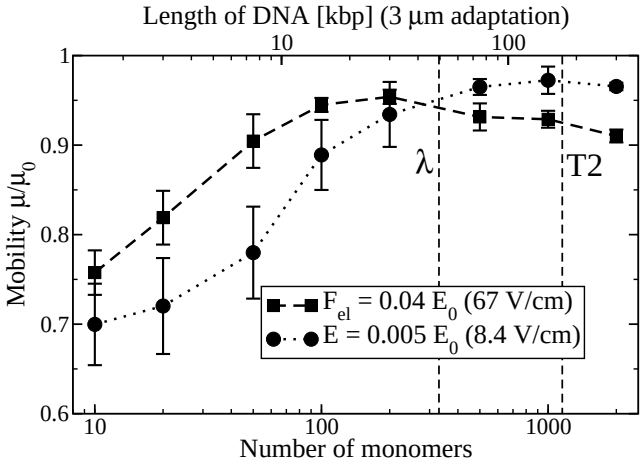

Figure 4.3: Mobility μ in the microchannel at low fields, $E = 0.005$ (8.4V/cm) and $0.04E_0$ (67V/cm). The vertical dashed lines indicate the length of λ- and T2-DNA, with adaptations for 3μm channel width setup. Experimentally, Duong *et al.* have reported $\mu/\mu_0 = 0.99 \pm 0.1$ for λ-DNA and $\mu/\mu_0 = 0.63 \pm 0.03$ for T2-DNA. [62]

Mobilities obtained from simulation are shown in figure 4.3 for $E = 0.005$, and $0.04E_0$. Also indicated is the corresponding chain length for the 3μm adaptation.

At low fields, $E = 0.0025E_0$,the simulation data indicates that the longer chains migrate faster than short chains, as expected from chapter 3 and reference [58]. This is in contrast to the observations presented by Duong *et al.* [62]. However, at such small fields it is very hard to determine mobilities experimentally, as the migration is dominated by diffusion, and the separation length is rather short. In my simulations, I am able to average over very long runs, and thus minimize diffusive effects.

For greater fields, $E = 0.04E_0$, however, the mobility exhibits a maximum at $N \approx 200$ monomers (28 kbp), and decreases for longer chains. This is in agreement with the observations presented by Duong *et al.*: they also observe that the longer $T2$-DNA migrates slower than the shorter λ-DNA. However, the decrease in mobility is not as pronounced as in experiment.

This result is gratifying, but it does not explain the sudden reversal of the migration

order. To understand this effect, even stronger fields are applied. The results of those investigations shall be described below.

4.5 Simulation at high electric Field

To understand the reversal in the chain length dependent mobility (Fig. 4.3), even more extreme conditions were investigated. The range of the electric field was extended up to E_0. Experimentally, it is difficult to operate at very high fields. Alternatively, table 4.1 suggests the use of larger devices, which have been used in experiments [63]. Using the adaptation for the 5μm channels, the electric field in simulation new covers the range up to 430V/cm.

4.5.1 Trajectories from Simulation

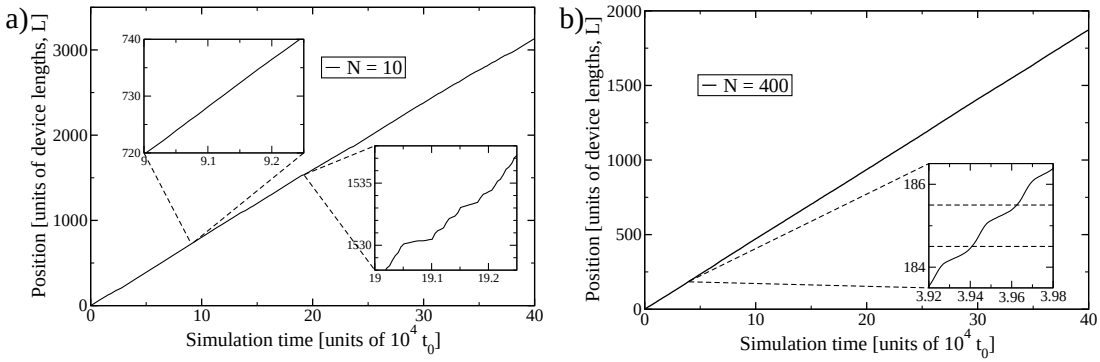

Figure 4.4: Trajectory of a) $N = 10$ (3.4 kbp), and b) $N = 400$ (140 kbp) monomers at $E = E_0(430\text{V/cm})$ in the structured microchannel. The dashed lines in the inset of $N = 400$ show the beginning of the narrow regions.

Trajectories obtained from simulation at $E = E_0$ (430V/cm) are given in figure 4.4 for chain lengths $N = 10$ (3.4 kbp) and $N = 400$ monomers (140 kbp). On long time scales, both trajectories are quite regular. The insets show the migration on short time scales.

In the case $N = 10$, trapping may still occur. This underlines the effect described in chapter 3.4.5, because the short chains exhibit a contour length that is shorter than the width of the narrow region, and thus may pass in any conformation. Furthermore, successive escape processes are correlated, even though the escape time is longer than the relaxation time. This effect comes from the diffusion during the passing of a single device unit. It allows for a mean quadratic drift of only $\approx 8.5\sigma$, which is smaller than the width of the narrow region.

For long chains ($N = 400$), the trajectory is very smooth. It does, however, exhibit a regular pattern, which can be seen in the inset. This effect is simply related to the fact

that the chain penetrates the deep region of the device. In that region, the electric field is low, which visibly affects the migration.

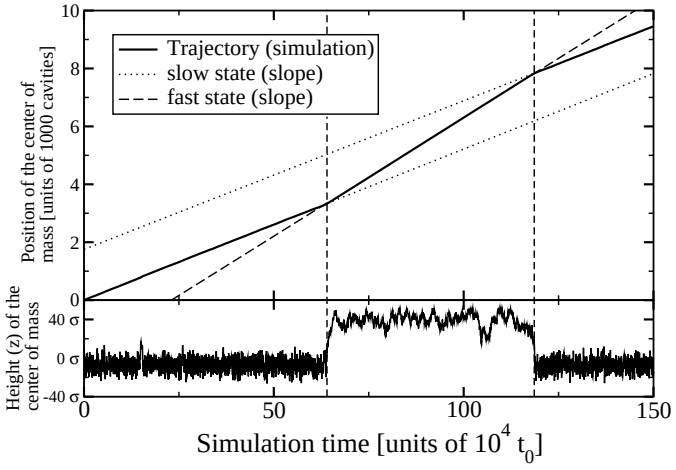

Figure 4.5: Trajectory of $N = 200$ monomers (70 kbp) at $E = E_0(430\text{V/cm})$ in the structured microchannel along the x-axis (top), and along the z-axis (bottom). The trajectory exhibits two distinct migration speeds, which can be related to the penetration depth of the wide regions. Thus two different migration states exist. The horizontal lines are just guides to the eye.

The situation is different for chains of medium length. A trajectory of $N = 200$ monomers (70 kbp) is shown in figure 4.5. The trajectory exhibits two distinct migration states, one of which is slow and the other is fast. Both migration states exhibit smooth trajectories on short time scales, and occasionally change into one another. Also shown is the penetration depth of the deep region during migration. Apparently, the migration speed is directly correlated to the penetration depth. If the chain enters the low-field area in the deep region ($z < 0$), it gets slowed down notably. Figure 4.5 also shows that the two migration states have long life times. In experiment, two migration states were found as well [63].

4.5.2 Snapshots

Snapshots obtained from simulation and experiment [63] are given in figure 4.6. The experimental snapshots show a time series taken with 40 ms time step. Both reveal that the two migration states are substantially different from each other.

In the fast state, the chains remain in the upper, homogeneous part of the field and are able to form a coil during migration. As the chains are permanently in the homogeneous part of the field, they migrate at a constant speed and do not lose time by passing low-field regions or getting stuck at walls. Thus the migration is fast.

The situation is different for the slow state. In this state, chains penetrate the wide regions of the device, and get slowed down by the reduced electric field in that area.

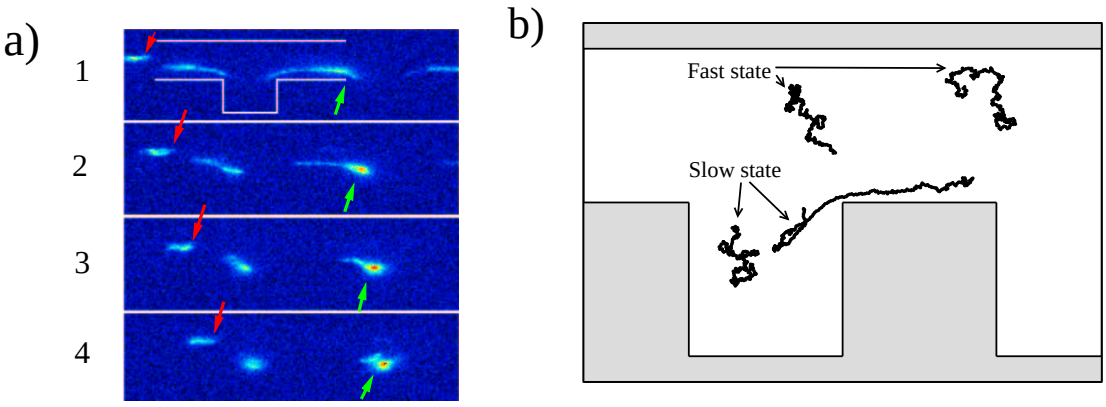

Figure 4.6: Snapshots obtained from (a) an experiment with T2-DNA [63] and (b) simulation with 200 beads at $E = E_0$. The experimental snapshots show a series taken with 40 ms time step (fast state shown by red arrows pointing down, slow state indicated by green arrows pointing up). Both snapshots show chains in the slow and the fast state. In the slow state, the chains penetrate the wide region, form a coil inside and stretch when passing into the narrow region. Chains in the fast state are also shown. Here the polymer remains permanently coiled in the homogeneous part of the field.

Inside the wide region, chains are able to form a coil. Note that a coiled conformation does not indicate that the chains are able to relax in this region, as the passing time is too short for long chains. When leaving the wide region, an initial loop enters the narrow region. Thus the chains get stretched, as they experience an inhomogeneous electric field. When leaving the narrow regions, the chains get pulled back into the wide regions considerably by the electric field lines.

4.5.3 Monomer Density Histograms

Monomer density histograms obtained from simulation are given in figure 4.7 for $N = 100$, 200, and 500 monomers at electric fields $E = 0.02$, 0.08, 0.5, and $1E_0$.

At low electric fields, $E = 0.02E_0$, all chains explore the full width of the continuous region. Short chains also explore parts of the wide regions, which is not penetrated by long chains.

At $E = 0.08E_0$, the situation is different. Whereas short chains still explore the full width of the continuous region, long chains are confined to a narrow region which enters the deep region. This confinement region compares well with the electric field lines (Fig. 4.1). Chains of intermediate length ($N = 200$ monomers) show an almost continuous depletion from the monomer density peak at the bottom of the narrow region to the top of that region and are thus some kind of intermediate of the densities exhibited for $N = 100$ and 500 monomers.

At even higher fields, $E = 0.5E_0$, even short chains exhibit a depletion zone between

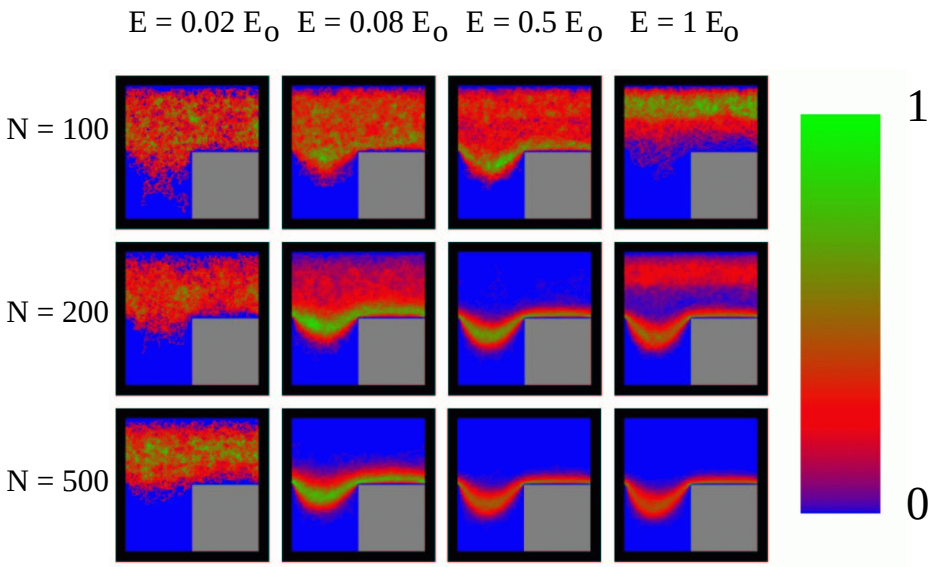

Figure 4.7: Monomer density histograms obtained from simulation for $N = 100$, 200, and 500 monomers at $E = 0.02$, 0.08, 0.5, and $1E_0$. For low fields ($E = 0.02E_0$), all chains explore all of the continuous region. At $E = 0.08E_0$, long chains ($N = 500$ monomers) occupy the slow state only, and at $E = 0.5E_0$ this is true for $N = 200$ monomers as well. At $E = E_0$, inertia comes into play, and shifts the transition region to longer chains. At $N = 200$ and $E = E_0$, the depletion zone between the two states can be identified nicely.

the two states, which is unfortunately hardly visible. This is caused by the fact that the chains still have a certain size. On the other hand, the density distribution of the center of mass is not very suitable as well: it may easily extend to outside of the device, if the chain is bend around a corner of the device (Fig. 4.6). Chains of intermediate length ($N = 200$ monomers) and longer ones are confined to the slow state.

At $E = E_0$, long chains still occupy the slow state only. Chains with a length in the range from $N = 120 - 250$ monomers alternatingly migrate in the fast and the slow state (Fig. 4.5), and long chains occupy the slow state more often (Fig. 4.9). Short chains ($N = 50 - 100$ monomers) migrate in the fast state only, and even shorter ones still explore the whole device. Surprisingly, chains of length $N = 200$ monomers now exhibit two distinct states again. Between the two states, a rather wide depletion zone is visible. At this field, short chains ($N = 100$ monomers) are confined to a region at the top of the continuous region. These effects are caused by the inertia of the model and will be discussed in chapters 4.5.4 and 4.6.3.

A monomer density histogram obtained from experiment [63] is given in figure 4.8. It

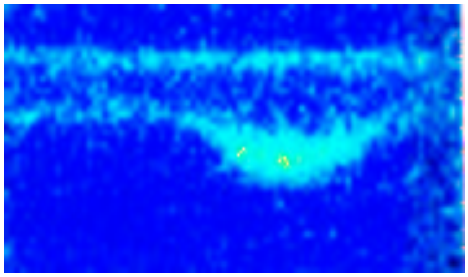

Figure 4.8: Monomer density histogram obtained from experiment at $E = 86\text{V/cm}$ ($0.20E_0$) for λ-DNA in 5μm channels [63]. Note the depletion zone between both states, which has been found in the simulations as well.

also shows a depletion zone between the two states, and therefore agrees to simulation data very well.

4.5.4 Population Density of the two States

Monomer density histograms such as those shown in figure 4.7 allow for determining the population densities of the two migration states. I chose the z-coordinate of the center of mass to determine the current migration state. If the center of mass is below 8σ, the chain is assumed to migrate in the slow state, and if it is above 20σ, it is assumed to be in the fast state. Furthermore, one of these conditions has to be fulfilled for at least $5 \cdot 10^6 \Delta_\text{t}$, otherwise the migration state is called "undefined". Figure 4.9 shows the average mobilities in the two states together with the total average of the mobility. Also shown in the lower panels are the population densities of the states.

The data proves that long chains are more likely to populate the slow state. Furthermore, the migration of short chains is dominated by diffusion, which is reflected by the amount of "undefined" population. At high fields ($E = E_0$), the transition is very sharp, and at low fields ($E = 0.04E_0$), it vanishes. However, a slight decrease of the mobility remains, thus explaining the reversal in the chain length dependence of the mobility.

The mobility as a function of the applied field for various chain lengths is shown in figure 4.10. Also shown are the mobilities of the fast and the slow states, as well as the population densities thereof.

For $N = 100$ monomers, the population density of the slow state rises with the electric field up to a critical value of $E \approx 0.35E_0$, and decreases for stronger fields. In that range, the amount of undefined data also decreases, which was roughly constant for lower fields. The overall mobility also exhibits a local minimum at $E = 0.35E_0$.

The situation is similar for $N = 200$ monomers and electric fields $E > 0.5E_0$. The amount of undefined data, however, has decreased considerably before. For electric fields $E > 0.5E_0$, the population densities suddenly start to reverse. This effect is probably an artifact of the simulation model, and is related to the mass m of a single monomer, and this effect is discussed in detail in chapter 4.6.3. For low fields, the chain mostly

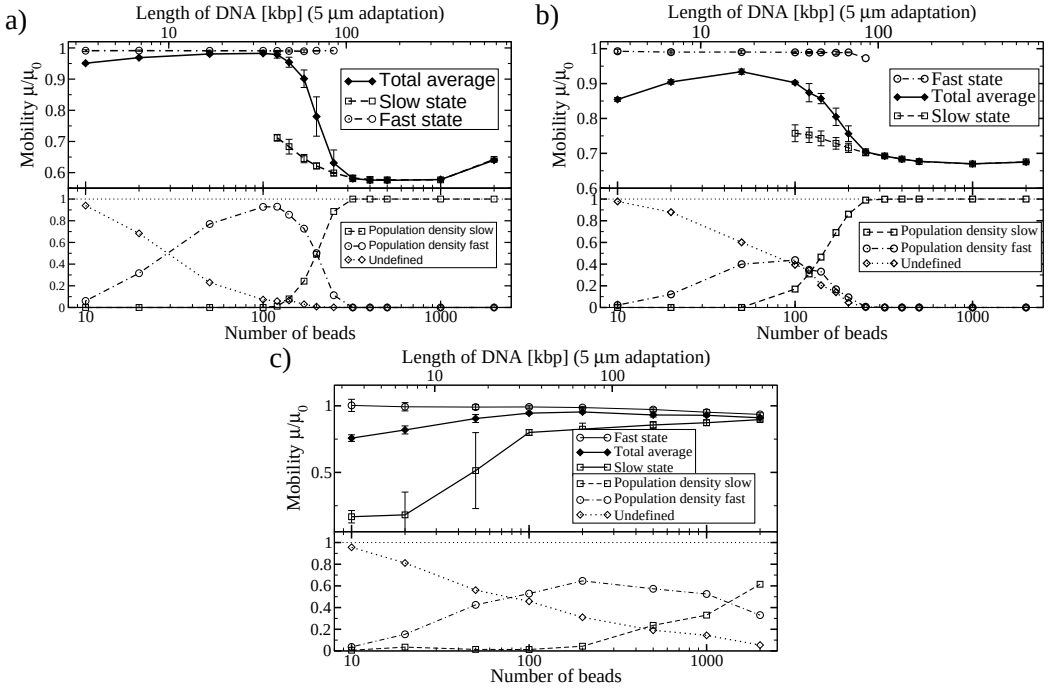

Figure 4.9: Mobility as a function of the chain length at $E = E_0$ (a), $E = 0.25E_0$ (b) and $E = 0.04\,E_0$ (c). Note the decrease in mobility at $N = 100 - 250$ (34 kbp – 85 kbp) at $E = E_0$, which is also reflected by the population densities given in the lower panel. For short chains, diffusion dominates the migration, and most of the time, it is not possible to assign unambiguously a single migration state. This is reflected by the amount of "undefined" population. The error bars of the slow and fast mobilities show the statistical error, based on the spread of the values. Note that the transition fades away when the electric field is decreased.

occupies the fast state, and at electric fields $E \gtrsim 0.11E_0$, the chain prefers to populate the slow state, before the population density is reversed.

In the case of $N = 500$ monomers, the chain occupies the fast state for $E \lesssim 0.05E_0$. This can simply be related to the fact that it is favored to occupy the large region on top, where the chain may form a coil. For greater fields, the favored migration state changes to the slow state, which is caused by the inhomogeneous field in the device and is discussed in chapter 4.6.1.

4.5.5 Infinitely deep Device

An important point that has been neglected in the matching of the experimental to the simulation parameters is the depth of the device. In experiment [62, 63], the depth is always 2.8μm, and thus comparable to the other device dimensions, and to the diameter

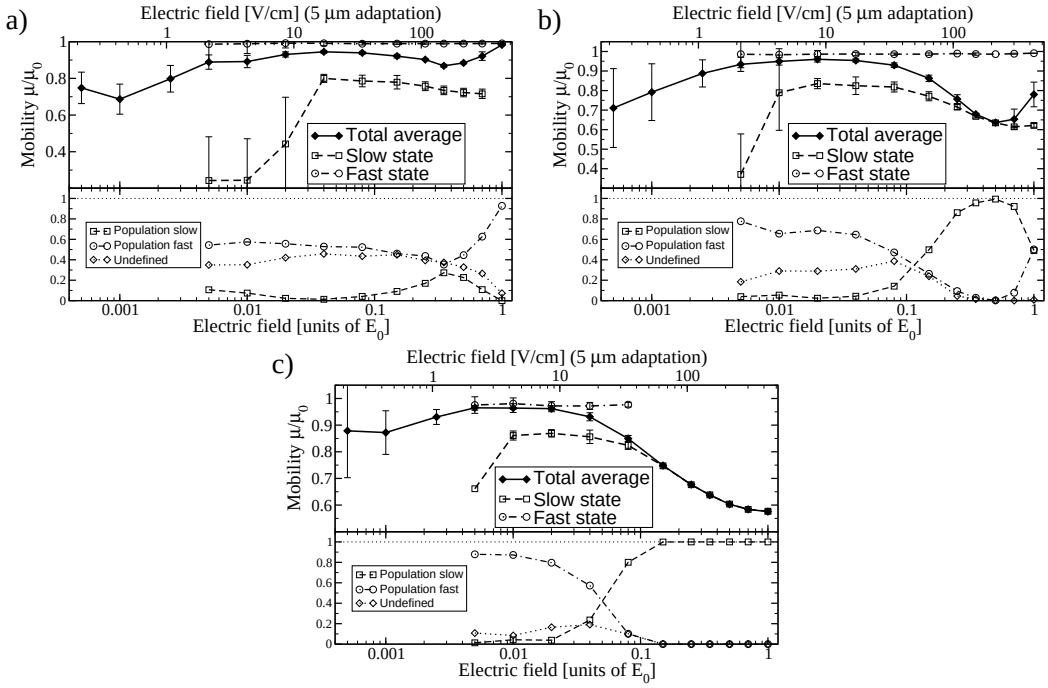

Figure 4.10: Mobility as a function of the electric field for $N = 100$ (a, 34 kbp), $N = 200$ (b, 68 kbp), and $N = 500$ (c, 170 kbp). Also shown are the mobilities in the slow and the fast state. Note the error of the latter is based on the statistical spread of the value. In the lower panel, the population densities of the two states are shown. For low fields or short chains, the assignment criterion may fail. The amount of unassigned migration data is marked by "undefined". Long chains tend to populate the slow state at high fields. For $N = 100$ and $N = 200$ monomers, the overall mobility and the population density exhibit a local minimum, which can be related to inertia effects (Ch. 4.6.3).

of the investigated DNA molecules. In simulation, the depth is set to 60σ, and *one* adaptation has been made to all *three* experimental setups (Ch. 4.3).

To test the influence of the device depth on the simulation results, simulations with infinite depth, or no walls in y-direction, were performed. Sample trajectories of $N = 100$, 200, and 500 monomers at $E = E_0$ are given in figure 4.11. Long chains migrate in the slow state, and short chains occupy the fast state. Chains of intermediate length migrate alternatingly in the fast and slow state. Apparently, the mobilities in the fast and slow state do not differ significantly, and the chain length dependent mobility is again mainly caused by the different population densities of the two states.

As the simulation data presented here is in very good agreement with the data obtained from the device with finite depth (Ch. 4.5.1), no further investigation is necessary.

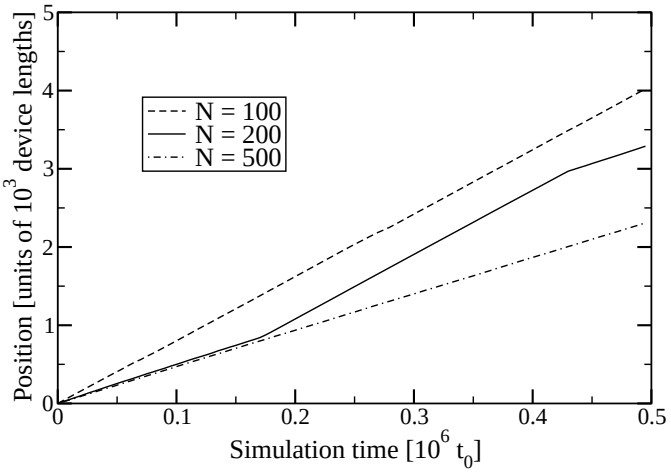

Figure 4.11: Trajectories in the infinitely deep microchannel for $N = 100$, 200, and 500 monomers at $E = E_0$. Short chains still migrate in the fast state, and long chains still occupy the slow state only. Chains of intermediate length migrate alternatingly in the fast and the slow state.

4.5.6 Meta-stable States

The population densities presented above suggest that it is impossible for long chains to populate the fast state at high fields. This suggestion implies that these chains, initially set up in the fast state, should change to the slow state in very short time.

Sample trajectories of these simulations are given in figure 4.12. A starting configuration of $N = 500$ monomers was taken from simulation at low field, where these chains populate the fast state (Fig. 4.10c). Unexpectedly, the long chains are able to populate the fast state for a very long time. The life time of the fast state turned out to be so large, that it was impossible to determine it in detail.

This data also explains why two migration states have been observed for T2-DNA (164 kbp, $\approx$ 480 monomers). As the experimental device exhibits a length of a few cm, it contains roughly roughly 10^3 trapping geometries. Thus the experimental setup hardly corresponds to the long-time limit investigated in my simulations. Furthermore, it is difficult to control the precise way how the chains enter the microchannel and thereby the initial migration state.

4.5.7 Parallelized Device

A device consisting of a single period in the z-direction is not convenient for separation, because it is unable to separate a large amount of samples in short time. Fortunately, an extension of the device to a parallel version, or array, of the device can be constructed very easily (Fig. 2.7).

Simulations of an array of geometrically structured microchannels have been carried

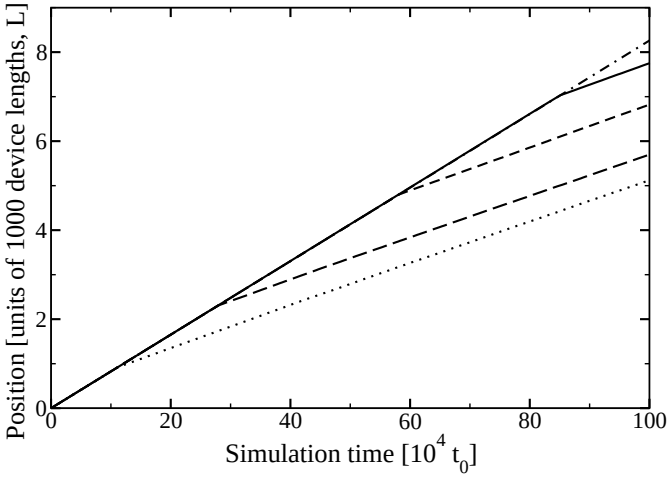

Figure 4.12: Trajectories of $N = 500$ at $E = E_0$, initially set up in the fast state. The chains are able to populate the fast state for a longer time, even though long-time averages predict the chains to be in the slow state only. The opposite transition has never been observed in the simulations.

out at electric fields $E = 0.001$, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, and $0.5E_0$. The simulated chain lengths were $N = 10$, 20, 50, 100, 200, 500, and 1000 monomers. Simulations were carried out for $7 \cdot 10^8 \Delta_t$, or $7 \cdot 10^6 t_0$ with inertia dynamics. The device geometry was left unchanged, $H = 60\sigma$. Note this implies that the width of the continuous region between the obstacles is now 120σ. Due to the symmetry of the device, the electric field distribution is equal to that used above.

A sample trajectory of $N = 200$ monomers at $E = 0.5E_0$ is shown in figure 4.13. Again, the trajectory exhibits two distinct migration speeds, which correlate with the penetration depth of the deep regions. The limits of the deep regions are shown by horizontal dotted lines, and are now on both sides of the continuous region in the middle. Thus my data proves that the two migration states also exist in an array device.

To check whether the migration is affected notably by the change to the array geometry, the mobilities were computed. Comparisons of the mobilities obtained from the array device and the single channel are shown as a function of the chain length, N, in figure 4.14. The agreement is excellent for all three electric fields E investigated. Note that the slight deviation for $E = 0.1/0.08E_0$ simply comes from the fact that two slightly different electric fields are compared.

Furthermore, the mobility as a function of the electric field has been investigated, and the results are shown in figure 4.15. As expected from the previous analysis, the results are in complete agreement with one another. For low fields, the separation becomes inefficient. This is caused by diffusion, which dominates the migration at low fields. For long chains at low fields ($N = 500$ at $E \leq 0.001E_0$), the chain migrates faster in the array. This is probably caused by the fact that the array device exhibits a wider continuous region than the single channel.

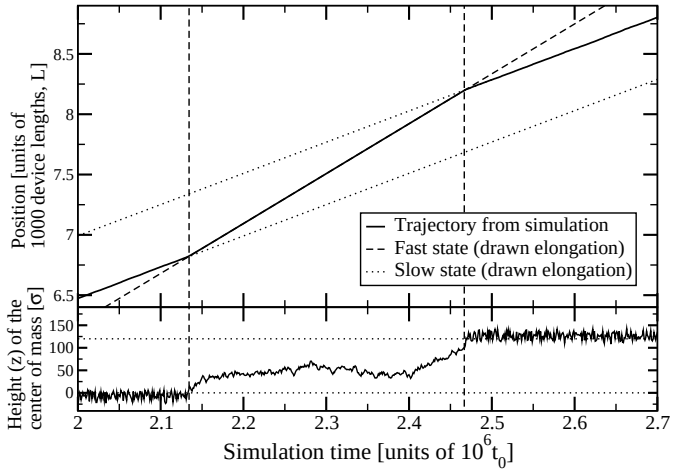

Figure 4.13: Trajectory of $N = 200$ monomers at $E = 0.5E_0$ in the parallel device. The trajectory in the upper panel exhibits two distinct migration states. Again, the migration speed can be related directly to the penetration depth of the wide regions in the lower panel. The limits of the continuous region are marked by dotted lines. Note that the wide regions are now on both sides of the continuous region.

All investigations presented indicate that the array device and the single channel exhibit exactly the same physics. This is gratifying, as it allows for an efficient separation device, and the results obtained for the single channel remain valid.

4.6 Transition Mechanism

To achieve a better understanding of the migration states and the mechanisms stabilizing them the transitions from the fast to the slow and from the slow to the fast state were investigated. The results obtained are discussed below.

4.6.1 Transition from the fast to the slow State

The simulation data presented in chapter 4.5.6 contains several transitions. A detailed analysis of one of these is described below.

To understand the mechanism, the changes of the conformation of the polymer chain during the transition was investigated. The radius of gyration was split into its components, and normalized with respect to its free-flow value. The results of this analysis, together with the actual height z of the polymer, are shown in figure 4.16.

At the beginning ($x = 0\sigma$), the polymer chain migrates in the fast state. However, the polymer is not able to form an unperturbed coil, which can be seen from the components of the radius of gyration. This is caused by the size of the polymer. In free flow, the radius of gyration for chains with $N = 500$ monomers is $R_g = 18.0 \pm 0.5\sigma$ [86]. Therefore,

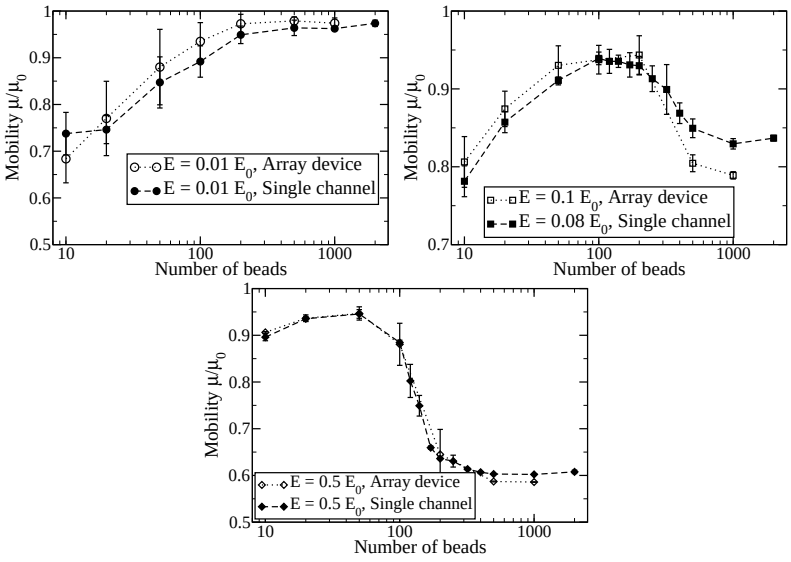

Figure 4.14: Comparison of the mobilities of the single channel to the array device as a function of the chain length, N, for electric fields $E = 0.01$, $0.08/0.1$, and $0.5E_0$. Note that at $E = 0.08/0.1E_0$, the mobilities slightly deviate from each other as the electric fields simulated differ. A comparison with the mobility depending on the electric field 4.15 yields good agreement for various chain lengths.

it is unable to fully extend in the y- and z-direction. The center of mass is slowly falling. A snapshot of this state is shown in figure 4.17a.

At some stage ($x \approx 7500\sigma$), the extension in the z-direction starts to rise slightly, whereas to other components are rather unperturbed. This indicates that an initial loop has started to penetrate the wide region.

A little later ($x \approx 10000\sigma$), the stretching in the z-direction has reached a considerable amount, and starts to affect the other extensions as well. A snapshot of this stage is shown in figure 4.17b. The initial loop has grown to a large amount of polymer, and is already being slowed down by the weak field in the wide region. As the polymer backbone is not parallel to the electric field, the inhomogeneous electric field induces a net down drift, which is also shown in the snapshot. Furthermore, the polymer is unable to withstand the inhomogeneity of the electric field, and starts to stretch the coil in the x-direction. This leads to a slight decay of the extension in the y-direction. The extension in the z-direction is still rising, but will soon start to decrease at $x \approx 11700\sigma$.

The polymer reaches a fully stretched conformation some time later, at $x \approx 12800\sigma$. At this point, the extensions in the y- and z-direction are minimal, and a snapshot is given in figure 4.17c. The leading monomers, which are still in the homogenous part of the field, are being pulled down very quick now.

Eventually ($x \approx 13000\sigma$), the leading monomer fails to enter the narrow region im-

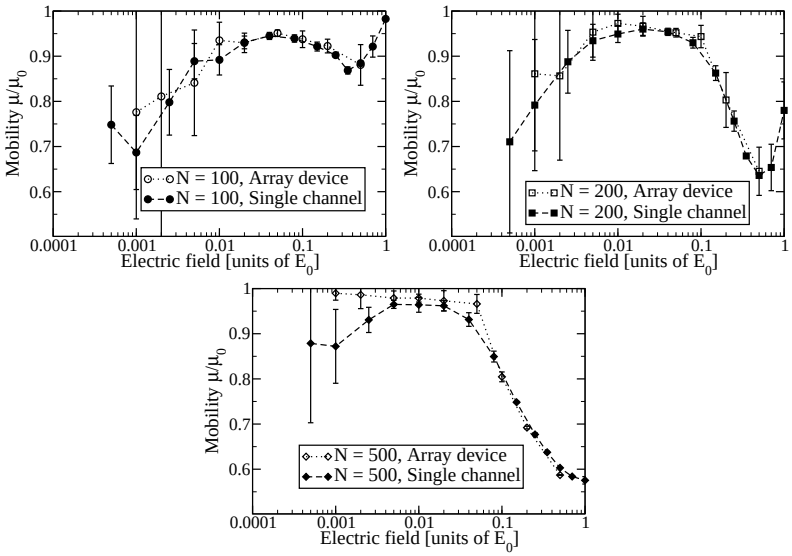

Figure 4.15: Comparison of the mobilities of the single channel to the array device as a function of the electric field, E, for chain lengths $N = 100$, 200, and 500. For long chains at low fields, the total width of the continuous region comes into play. This is wider for the array device, hence the increased mobility.

mediately, and it gets stuck in the wide region. Then, the whole polymer chain, which was stretched completely, collapses in a single device unit, as shown in figure 4.17d. At this point, the extensions in the y- and z-direction start to increase again, as the stress on the polymer chain is reduced. In the x-direction, the electric field still stretches the polymer considerably.

To summarize, the slow state exists because the electric field is inhomogeneous, and is able to stretch the polymer chain to a certain amount. This will be discussed in chapter 4.6.5.

4.6.2 Inertia dynamics

Snapshots of the transition from the slow to the fast state are not as easy to understand as those of the opposite transition described above. Simulations of a single monomer in the limit $T \to 0$ reveal an artifact of the simulation model with finite mass. The result of that simulation is shown in figure 4.18.

When migrating around the corners of the device, the monomer migrates at high speed. Therefore, it is unable to follow the electric field lines exactly. Thus, there is an up drift induced by the inhomogeneous electric field. On the other hand, the velocity in the wide regions is rather low, and the down drift induced by the finite mass is much smaller than the up drift at the corners. This results in a net up drift of a single monomer, and short chains, in strong electric fields.

The electrophoretic relaxation time τ_{el} in my model with inertia dynamics is $1t_0$,

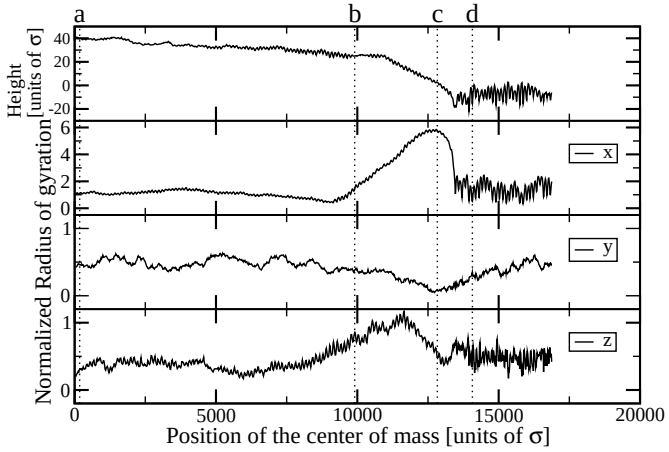

Figure 4.16: Radii of gyration during the transition from the fast to the slow state of $N = 500$ monomers, together with the height z of the center of mass. For a better understanding, the radius of gyration has been split into its components, which are given in the insets, and has also been normalized with respect to the free-flow expectation value. The positions of the snapshots presented in figure 4.17 are shown by dotted lines.

which corresponds to $\sim 10^{-4}$sec in experiment. Grossmann has reported $\tau_{\mathrm{el}} = 10^{-9} - 10^{-12}$sec [42]. Thus the up drift induced by the electric field is an artifact of the inertia dynamics.

4.6.3 Overdamped Dynamics

Figure 4.18 also reveals that for overdamped dynamics (Eq. 2.14) even a single particle does not drift upwards except for numerical errors. The time step used for overdamped dynamics is $\Delta_{\mathrm{t}} = 10^{-4} t_0$. Thus numeric errors are insignificant, which is proven in figure 4.18. To prove that the two migration states are not an artifact of the simulation model, chains in the overdamped limit $m \to 0$ were simulated. Due to the reduced time step, only short chains with $N \leq 100$ were investigated. Details of the simulations are given in chapter 4.2.

Sample trajectories for $N = 100$ monomers at $E = 0.5E_0$ for different starting configurations are shown in figure 4.19. The system still exhibits two distinct migration speeds, and the transition still occurs in both directions. Thus the two migration speeds are not an artifact of the simulation model. A detailed analysis of the data is not possible, because the time step Δ_{t} was reduced by a factor of 100 in order to integrate the harmonic springs properly.

Examining the mobility as a function of the electric field, one finds that chains from $N = 100$ up to $N = 250$ exhibit a minimal mobility at high fields (Fig. 4.10). The associated electric field of the minimal mobilities turned out to be $E_{\mathrm{min}} = 0.35E_0$ for

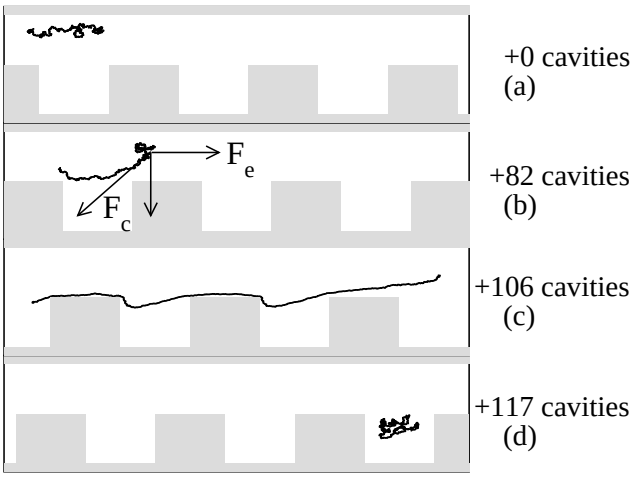

Figure 4.17: Transition snapshots from the fast to the slow state. The snapshots are shifted by the number of device lengths given on the right. (b) also shows a force parallelogram indicating the generation of a net down drift: the force F_e and the spring force F_c on the leading monomers add up to a total force that points downwards.

$N = 100$, $E_{min} = 0.7E_0$ for $N = 250$, and $E_{min} = 0.5E_0$ for all intermediate chain lengths. Shorter chains exhibit monotonous increase of the mobility as a function of the applied field, and long chains show a monotonous decrease in the two-state region of the electric field. The electric field, at which the field dependent mobility reaches a minimum, can be taken as an estimate of the electric field strength at which inertia effects come into account. Therefore, electric fields were limited up to $E_{max} = 0.5E_0$ in the case of pulsed fields (Ch. 5).

To conclude, the inertia of the monomers does suppress the transition to the slow state for short chains at high fields and therefore shift the transition region, but it does not explain the two migration states.

4.6.4 Transition from the slow to the fast State

As shown above, the transition regime at high fields is related to an artifact of the simulation model, but it is not an artifact of the model itself. Unfortunately, the snapshots of the transition from the slow to the fast state do not reveal the transition mechanism as easily.

One possible explanation is the fact that stretching of a polymer chain costs entropic free energy [104]. Especially when migrating around the corners, the polymer is forced into a fully stretched conformation. Thus the urge of the polymer to form a coil induces a drift upwards into the homogeneous field region. This effect is quite similar to the trapping of chains in the wide region of the device (Ch. 3.4.5): the polymer favors

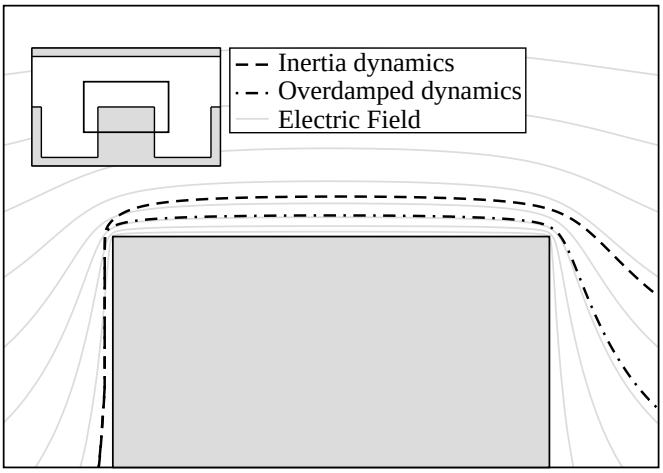

Figure 4.18: Up drift induced by inertia dynamics. At the corners, the electric field is very strong, and single monomers and short chains are unable to follow the electric field exactly. Also shown is the trajectory of an overdamped monomer, which is able to follow the electric field lines exactly.

to be in a homogeneous field area, as this area does not exert stress on the polymer conformation.

Another conceivable mechanism is that the chain migration is affected by diffusion. However, when migrating along the wall in the narrow region, diffusion is possible in only one direction, towards the homogeneous region. This also leads to a net up drift. This effect can also be seen in the monomer density histograms (Fig 4.7): at the top of the channel exists such a depletion zone as well.

A detailed analysis of the transition is difficult, because both transitions are triggered by diffusion. Thus the transition mechanism from the slow to the fast state remains an open problem.

4.6.5 Crossover Chain Length

As already seen in chapter 4.6.1, chains need to be stretched above a critical limit to remain in the slow state. The stretching itself is induced by the inhomogeneous electric field, and thus works against the entropic spring force of the polymer. The mean elongation $\langle r \rangle$ of a self-avoiding polymer subject to an external force f, which pulls on both ends, is given by [104]

$$\langle r \rangle \propto Na(fa/k_{\mathrm{B}}T)^{2/3} \tag{4.1}$$

for strongly stretched polymers and

$$\langle r \rangle \propto \frac{a^2 N^{2\nu}}{k_{\mathrm{B}}T} f \tag{4.2}$$

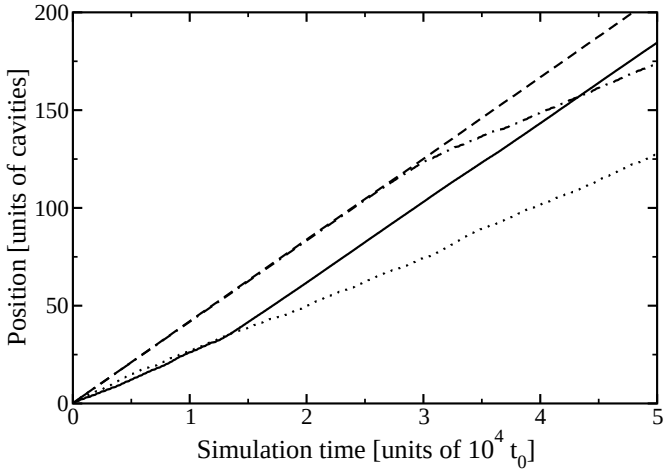

Figure 4.19: Trajectories of $N = 100$ monomers with overdamped dynamics (Ch. 2.2.2) at $E = 0.5E_0$, for different starting configurations. Two distinct migration states still exist.

for weakly stretched polymers, where $a \approx 0.5\sigma$ is the proportionality constant in equation 2.27. In this case, the force acts on all monomers, and is not constant along the polymer chain. Nevertheless, I will use $f \propto E$, together with equations 4.1 and 4.2, to roughly estimate the amount of stretching exerted by the inhomogeneous electric field onto the chain.

It is not yet clear how much the chain has to be stretched in order to remain in the slow state. Generally, there are two possibilities: the chain has either to be stretched to an amount determined by its own size, or it has to be stretched to an amount determined by the size of the device.

Assuming that it is necessary to stretch the chain to a length proportional to its own size, both equations 4.1 and 4.2 yield $NE^{5/3}$ to be a constant. On the other hand, a length proportional to the size of the device represents a strong stretching (Eq. 4.1), and this estimate predicts $NE^{2/3}$ to be constant.

I use the chain length N_c, at which both states are equally populated, as a reference. Both predictions are tested against simulation data in table 4.2. Obviously, the prediction $N_c E^{5/3}$ is not constant, as it raises monotonously with the electric field, E. On the other hand, $N_c E^{2/3}$ is almost constant for electric fields $E \leq 0.5E_0$, but rises for stronger fields. However, the simulation data exhibits inertia related effects for $E \gtrsim 0.5E_0$, which is shown in chapters 4.5.3, 4.5.4, 4.6.2, and 4.6.3. For lower fields, the rule $N_c E^{2/3} \approx 50E_0^{2/3}$ is roughly fulfilled. Assuming that this rule is valid, simulations of the overdamped dynamics should lead to a crossover chain length of

$$N_c \approx 50 \tag{4.3}$$

at $E = E_0$. Sample trajectories of $N = 50$ monomers at $E = E_0$ are shown in figure 4.20.

E/E_0	N_c	$N_c \cdot (E/E_0)^{5/3}$	$N_c \cdot (E/E_0)^{2/3}$
0.08	290 ± 25	4.3 ± 0.4	55 ± 5
0.15	174 ± 20	7.4 ± 0.9	50 ± 6
0.25	124 ± 15	12.3 ± 1.5	50 ± 6
0.35	120 ± 15	20.9 ± 2.6	60 ± 8
0.50	106 ± 10	33.4 ± 3.2	67 ± 6
0.70	125 ± 10	71.7 ± 5.7	103 ± 8
1.00	200 ± 5	200.0 ± 5.0	200 ± 5

Table 4.2: Crossover lengths N_c for various electric fields. Errors of the crossover lengths are estimates of the population density histograms.

The chains exhibit two distinct migration states. Unfortunately, the data is too poor to determine a crossover length N_c.

Nevertheless, my data suggests that the crossover chain length depends on the device length, and not on the chain length itself. This result also gives a possible explanation why the transition mechanism fails at low fields: equation 4.3 predicts the crossover chain length at $E = 0.04E_0$ to be $N_c \approx 500$. The simulation data, however, does not support this finding (Fig. 4.9). On the other hand, chains of $N = 500$ monomers exhibit an end-to-end distance of roughly 46σ. Hence, the polymer is not able to form a coiled conformation in the homogenous field region, and the transition mechanism fails.

The results presented here also suggests strategies for enhancing/ suppressing the transition mechanism presented here: the two-state behavior can be promoted by increasing the homogeneous field region at the top of the device. This can be easily achieved by increasing the channel height, c, in figure 4.1.

4.7 Conclusions

To summarize, I have investigated the migration of DNA by a coarse-grained model with a Brownian dynamics simulation. My simulations reproduce the migration order of λ- and T2-DNA reported by Duong *et al.* [62]. In particular, my simulation data explains why the chain length dependent mobility is suddenly reversed. In channels such as shown in figure 4.1, and at moderate electric fields, the longer T2-DNA molecule migrates slower than the shorter λ-DNA molecule.

This behavior, which is opposite to that expected at lower fields, is a signature of a high-field non-equilibrium bistability. At very high fields (or in larger structures, *cf.* table 4.1), chains migrate with two distinct migration speeds, indicating two different states of migration. This two-state behavior has been observed in experiments as well [63]. Factors which stabilize the two states have been discussed, and thus strategies to construct separation devices, which either promote or suppress this behavior, can be proposed.

A comparison of the experimental to the simulation data yields that the data obtained

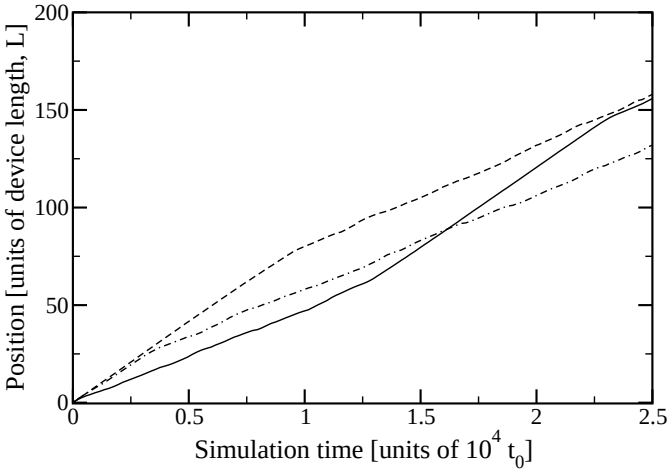

Figure 4.20: Trajectories of $N = 50$ monomers with overdamped dynamics (Ch. 2.2.2) at $E = E_0$, for different starting configurations. Two distinct migration are clearly visible.

in experiment and from simulations agree very well. This implies that my simulation model, which disregards electrostatic and hydrodynamic interactions, and assumes a simple electroosmotic flow, nevertheless captures the essential physics of the migration of DNA in an external field with geometrically structured microchannels. However, the interplay of DNA motion and buffer flow is non-trivial and may be neglected for low fields outside the Debye layer only. As reported in reference [63], the electrophoretic mobility is of the same order of magnitude as the electroosmotic mobility. This should give rise to induces interesting flow patterns and non-trivial behavior in the case of DNA migrating along the Debye layer, or at high fields. Such flows have already been investigated for an inhomogeneous space charge density [107]. Moreover, electrostatic and hydrodynamic interactions are not entirely screened in the presence of geometric barriers. This leads to additional effects [101]. In order to investigate such phenomena, systematic studies are necessary, and efficient new separation techniques need to be developed as well.

An array of microchannels such as those described here may easily be constructed. In such a device, a large amount of sample DNA can be separated in parallel, and the separation of large amounts of sample DNA can be accelerated considerably. As shown here, the simulation data shows that the data obtained for a single channel is the same as for the array device, given a sufficient electric field. In this case, the data can be transfered directly, and the effect of the covering walls are indeed neglectable.

From a practical point of view, these results have two implications:

One the one hand, the results show that the migration of DNA may be surprisingly complex. DNA may exhibit non-monotonic length dependence of the mobility, and even bistabilities during migration. These effects are non-trivial and must be considered during development of new separation devices, and a thorough theoretical analysis is recommended.

On the other hand, the observed two-state behavior might be exploited successfully when a time dependent electric field is applied. My data suggests that the mobilities of the fast and the slow state differ by a factor of almost 2 (Fig. 4.9 and 4.10), whereas the dependence of the mobility in the two states is rather weak. Furthermore, the crossover chain length N_c, at which both states are equally populated, may be easily adjusted by changing the size of the device. However, in order to benefit from this two-state migration behavior, the switching times between the two states need to be reduced considerably.

In the next chapter, I will investigate the effects of a time dependent electric field in the array device.

5 Pulsed electric Field

5.1 Introduction

The latest separation devices do not operate with a static geometry or an AC field alone, but on a combination of an AC field combined with a specific geometry.

Recently, Austin's group has presented a device consisting of a hexagonal array of posts [35, 40]. Electric fields and the direction thereof vary with time. With a proper setup, one can achieve that the field distribution around each post is the same inside the twodimensional device for any average field direction [108]. Note that does not imply that the local electric field itself is homogeneous.

Another ratchet device has been presented by Huang *et al.* [109], and the influence of the molecular size on the fractionation was investigated. Here, rectangular obstacles are placed with an orientation off the direction of the electric field. Migration of particles according to electric field lines is reported, and "memory" effects induced by the particles are discussed. Particles with a size comparable to the device dimensions lose their conformation memory when migrating through the barriers. This can be compared to the slow state described in chapter 4.5. The resolution reported of this device is 3 times better than Brownian ratchet arrays [110].

In this chapter, I will investigate another effect that should be applicable as a ratchet mechanism: the two-state migration presented in chapter 4. According to my results and those found in experiment [63], polymer chains populate both states depending on the electric field applied. With a proper setup, it should be possible to force the chain into the slow state with a strong field, and favor the fast state with a weak field. As the mobilities in these states differ by a factor of almost 2, a net drift in the direction of the weak field should occur. On the other hand, the transition rates between the two states are very low, indicating that the electric fields applied need to switched very slow.

This chapter is organized as follows: the setup of the simulations is described in chapter 5.2. The investigation is performed in two steps. First, I will investigate the influence of a time-symmetric electric field with mean zero in chapter 5.3, and compare the results to those expected from those presented in chapter 4. In a second step, I apply an asymmetric field with mean zero (Ch. 5.4) and investigate the minimum switching time necessary to exploit the ratchet effect.

5.2 Simulation Setup

To exploit the two-state migration behavior, the device dimensions of chapter 4 were used. In that simulation, I set $H = 60\mu$m, which is also the depth of the device in the

lateral direction. Furthermore, the extension to an array device was used. A comparison of array data to data of a single microchannel is given in chapter 4.5.7. For an efficient simulation, I used the dynamics with inertia, and electric fields were limited to a maximum field strength of $E_{\max} = 0.5E_0$ to avoid artifacts caused by the dynamics with inertia (Ch. 4.6.5). As the adaptation to experimental values presented in chapter 4.3 remains valid, no new matching of parameters was performed.

Compared to a static electric field, a pulsed field offers three additional parameters (Ch. 2.9.1). This implies that looking for a proper parameter set can be very time consuming. To reduce the number of parameters, initial investigations were restricted to a mean-zero, time-symmetric electric field. Electric fields applied covered the values $E = 0.01$, 0.02, 0.05, 0.1, 0.2, and $0.5E_0$, and switching times of the electric field were adjusted so that in free flow, the chains would have migrated $\Delta_x = \{2.5, 5, 10\}(l_\mathrm{b} + l_\mathrm{t})$. Figure 2.6 shows that the total length L of the device is the sum of the length of the wide region, l_b, and the length of the narrow region, l_t. This implies that the switching time $t_\mathrm{sw} = \Delta_x/E\mu_0$ varied from $t_\mathrm{sw} = 24 \cdot 10^3 - 12 \cdot 10^6\Delta_\mathrm{t}$. Note that the switching time depends on the electric field. If the switching time was independent of the electric field, the chains would either not have migrated considerably, or over very long distances, before the field is reversed. Simulations were carried out over $9 \cdot 10^9\Delta_\mathrm{t}$ for chain lengths of $N = 10$, 20, 50, 100, 200, 500, and 1000 monomers.

In a second analysis, electric fields with mean-zero, but with a broken time symmetry were applied to the device. As the field exhibits a mean-zero value, the field consists of two different pulses: a short and strong, and a weak and long one. In my simulations, the strong pulse is always directed in the positive x-direction. I investigated two cases (Fig. 2.9): $\Delta t_1/T = 0.2$ and $\Delta t_1/T = 0.3$, with t_1 the application time of the strong pulse, and the period T. Note that $\Delta t_1/T = 1/2$ leads to the time-symmetric field described above. Application of $E_1\Delta t_1 = -E_2\Delta t_2$ yields $E_1/E_2 = -1/4$ and $E_1/E_2 = -3/7$, respectively. Electric fields applied cover the range $E_1 = 0.1$, 0.2, and $0.5E_0$. Thus the electric fields cover the range of the two-state behavior (Ch. 4). Switching times were chosen so that the chains are able to migrate distances $\Delta_x = \{1, 2, 3, 5\}L$ in free flow. This implies that the strong pulse is applied for $\Delta t_1 = 24 \cdot 10^3 - 1.2 \cdot 10^6\Delta_\mathrm{t}$, and the total period of the electric field ranges from $T = 80 \cdot 10^3 - 6 \cdot 10^6\Delta_\mathrm{t}$. Simulations were carried out for $6 \cdot 10^8\Delta_\mathrm{t}$ for migration distances $\Delta_x = \{1, 2, 3\}L$, and for $10^9\Delta_\mathrm{t}$ in the case of $\Delta_x = 5L$. Investigated chain lengths were $N = 10$, 20, 50, 100, 200, 500, and 1000 monomers.

Note that the simplification of an electroosmotic flow similar to the electric field outside the Debye layer at the walls (Ch. 2.10.2, Ref. [99, 100]) is only true for a static electric field. However, the shortest time of application of the strong pulse is $24 \cdot 10^3\Delta_\mathrm{t} = 240t_0$. The adaptation given in table 4.1 for the 5μm structures yields that 1sec $\equiv 10^4 t_0$. Thus electric fields are applied for at least 24msec in experiment. Experiment reports indicate that electroosmotic flow establishes to its steady state behavior in about 100μsec [45, 26]. These considerations clearly show that the assumption of an electroosmotic flow similar to the electric field can still be justified.

5.3 Time-symmetric pulsed electric Field

5.3.1 Trajectories from Simulation

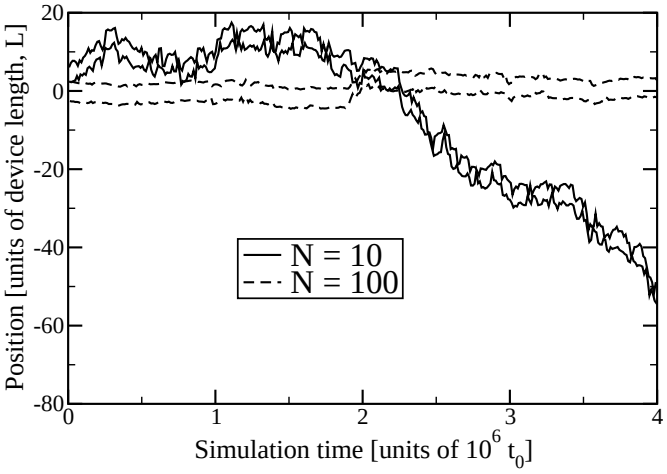

Figure 5.1: Position of the center of mass at the field reversal of chains with $N = 10$ and 100 monomers in the pulsed time-symmetric electric field at $E_1 = 0.1E_0$. As the field exerts a time-averaged mean-zero force, the chains exhibit diffusive effects on long time scales only. For short time scales, however, the migration of the chain is controlled by the electric field. Here, one single pulse of the electric field allows for a free-flow migration distance of $\Delta_x = 5L$.

Sample trajectories of $N = 10$ and 100 monomers in an electric field $E_1 = -E_2 = 0.1E_0$ at pulse times of $t_{\mathrm{sw}} = 6 \cdot 10^3 t_0$ are shown in figure 5.1. This implies that chains are able to migrate $5L$ in free flow during one single pulse. As the migration is dominated by the forward and backward movements of the electric fields, only the positions of the center of mass at the reversal of the field are shown. Thus the trajectory oscillates between the two lines shown. Apart from the direct migration induced by the electric field pulses, the chains still exhibit diffusive motion, which is much stronger in the case of $N = 10$ monomers.

The trajectories differ from those shown above if the chain exhibits two states of migration, as described in chapter 4.5. A trajectory of $N = 200$ monomers at $E_1 = -E_2 = 0.5E_0$, combined with the penetration depth of the wide region is shown in figure 5.2. If the chain is in the continuous region of the device, the deviation from the free-flow migration behavior is small and can be related to free-flow diffusion. This situation is changed considerably if the chains penetrate the wide regions. In this case, the migration is influenced by the device walls, and the inhomogeneous electric field. This leads to a reduced drift during a single pulse, which is described below. Furthermore, the diffusive motion in field direction is enhanced, which is apparent from the trajectory.

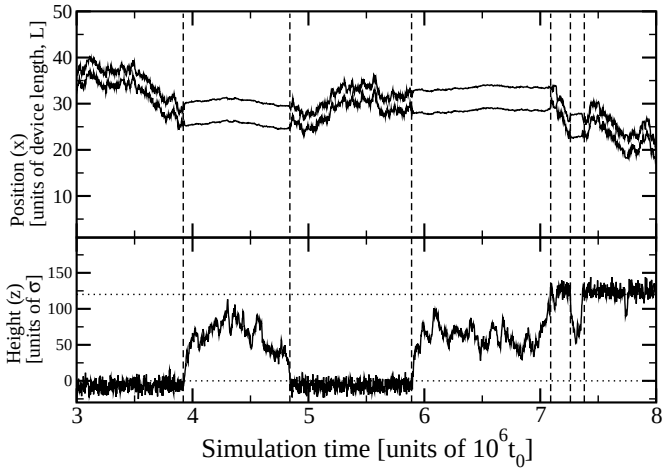

Figure 5.2: Trajectory of $N = 200$ monomers at $E = 0.5E_0$ in the time-symmetric pulsed electric field with $t_{\mathrm{sw}} = 5L/E_0\mu_0$, marked by the positions at field reversal. Two distinct states of forward and backward migration are visible. As in chapter 4.5, these are correlated to the level of penetration of the wide region shown in the lower panel. The horizontal dotted lines indicate the borders of the continuous region, and the vertical dashed lines are just guide to the eye.

5.3.2 Drift Histograms

Trajectories oscillate between the points when the electric field is reversed, and exhibit mean zero total drift. However, the change of position during one single (forward or backward) electric pulse, Δ_{x}, is influenced by the parameters of the pulse, like switching time t_{sw}, and E_1. The drift lengths of the trajectory presented in figure 5.2 are shown in figure 5.3. As was already visible before, the drift length does depend on the penetration depth of the wide region. If the chain is in the continuous region of the device, the drift length roughly corresponds to the free-flow expectation value $\Delta_x = 600\sigma$, and penetration of the wide region results in a reduced drift length per pulse. The amount of reduction depends on the interaction with the walls and the electric field along the migration path very sensitively, which explains why the drift length is spread over a wide range.

The probability density distribution of the drift lengths of chains with $N = 100$, 200, and 500 monomers is shown in figure 5.4.

For $N = 100$, only one peak exists, and it is roughly located at the free-flow expectation value $\Delta_{x,\mathrm{free}} = 600\sigma$. For $N = 500$ monomers, a single peak at $\Delta_x \approx 360\sigma$ exists.

However, at $N = 200$ monomers, two distinct peaks are visible. The peak at $\Delta_x = 600\sigma$ corresponds to the free-flow expectation value. In free flow, the width of that peak can be computed by equations 2.11 and 2.26:

$$\left\langle (r_x(t) - r_x(0))^2 \right\rangle = 2Dt. \tag{5.1}$$

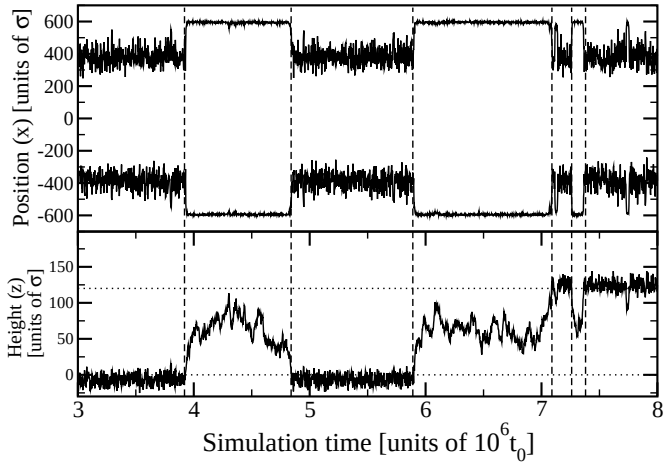

Figure 5.3: Drift lengths of $N = 200$ monomers at $E = 0.5E_0$ in the time-symmetric pulsed electric fieldwith pulse length $t_{\mathrm{sw}} = 5L/E_0\mu_0$. Inside the continuous region, the drift length during one single pulse of the electric field agrees very well with the free-flow prediction. If the chain penetrates the wide region, the drift length is reduced notably.

Note that in this case, only one direction, x, is considered. Substitution of $N = 200$ monomers and $t = 1200t_0$ yields a mean quadratic offset from the expectation value of

$$\left\langle (\Delta_x - \langle \Delta_x \rangle)^2 \right\rangle = 12\sigma^2. \tag{5.2}$$

Here, the peak width is apparently comparable to the theoretical prediction. Unfortunately, the available data does not allow a thorough analysis.

The other peak is related to the slow state described in chapter 4.5, and is located at $\Delta_x \sim 365 - 370\sigma$. A comparison with the two-state mobility data shown in figure 4.10 yields a mobility of $\mu/\mu_0 = 0.635 \pm 0.003$. This leads to a mean drift length of $\langle \Delta_x \rangle = (381 \pm 2)\sigma$, which is slightly deviates from the location of the maximum found here. However, a closer inspection reveals that

$$\langle \Delta_x \rangle = \int \mathrm{d}x \; \Delta_x P(\Delta_x). \tag{5.3}$$

Here, the probability distribution $P(\Delta_x)$ is not symmetrical around the maximum. Performing the integration shown above, the expectation value of the drift length is $\langle \Delta_x \rangle \approx 390\sigma$, where the data with $\Delta_x > 550\sigma$ has been cut off. This agrees very well with the prediction of the slow state.

To summarize, these data are almost identical to that obtained from the static electric field (Ch. 4.5.4): Short chains favor the fast state, long chains prefer the slow state, and chains of intermediate length migrate alternatingly in the fast and the slow state.

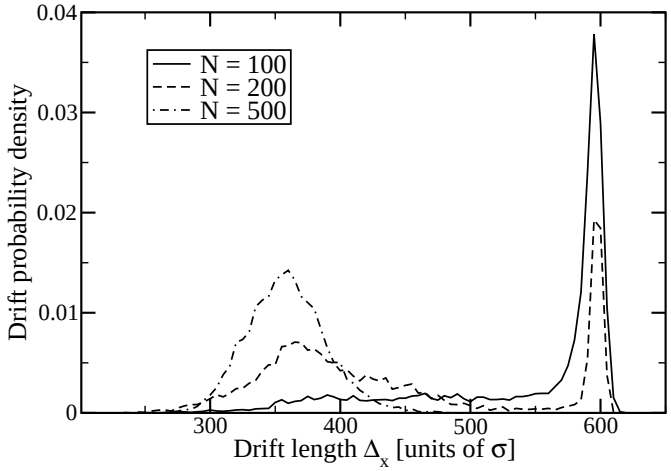

Figure 5.4: Drift length probability density of $N = 100$, 200, and 500 monomers at $E = 0.5E_0$ in the time-symmetric pulsed electric field with $t_{\mathrm{sw}} = 5L/E_0\mu_0$. The probability distribution exhibits two distinct peaks, which can easily be related to the penetration depth of the wide region.

5.3.3 Drift Length

Mean drift lengths $\langle \Delta_x \rangle$ at $E_1 = 0.2E_0$ as a function of the chain length, N, are shown for switching times $t_{\mathrm{sw}} = 1500$, 3000, and $6000t_0$. For a better comparison, all drift lengths have been normalized to the free-flow drift length, *i. e.* a value equal to unity indicates that the chain has not been influenced by the electric field or the walls at all. Here, the free-flow drift length is represented by $t_{\mathrm{sw}}E_1\mu_0 = 2.5$, 5, and $10L$. These drift lengths have also been compared to those expected from mobilities in a static electric field, which are indicated by dashed lines.

Apparently, the data agree if the chain is either short, or the switching rate of the electric field is low. One convincing explanation is that short chains are able to adapt to the steady-state behavior very quickly, as these exhibit both a short relaxation time τ_{R} (Ch. 2.7.1), and a large diffusion constant, D (Eq. 2.26). On the other hand, long chains need much time to adapt to a new electric field, as these have a long relaxation time and a low diffusion constant, D. However, as the pulses are symmetrical, the migration states in both forward and backward pulses are identical. The reason why long chains obviously need time to adapt to the reversed field, can be seen in figure 4.6: in the slow state, a chain consists of two parts. One part of the chain is coiled in the wide region, and the other is stretched into the narrow region. Thus the chains exhibit an orientation, and need to adapt to a new electric field direction, even if it is simply reversed. However, this process is much faster than a full relaxation of the conformation, which is $\tau_{\mathrm{R}} \approx 5200t_0$ for $N = 200$ monomers.

Mean drift lengths $\langle \Delta_x \rangle$ of chains with $N = 100$, 200, and 500 monomers as a function of the electric field E_1 are shown in figure 5.6 for a switching time $t_{\mathrm{sw}} = 2.5L/E_1\mu_0$. As

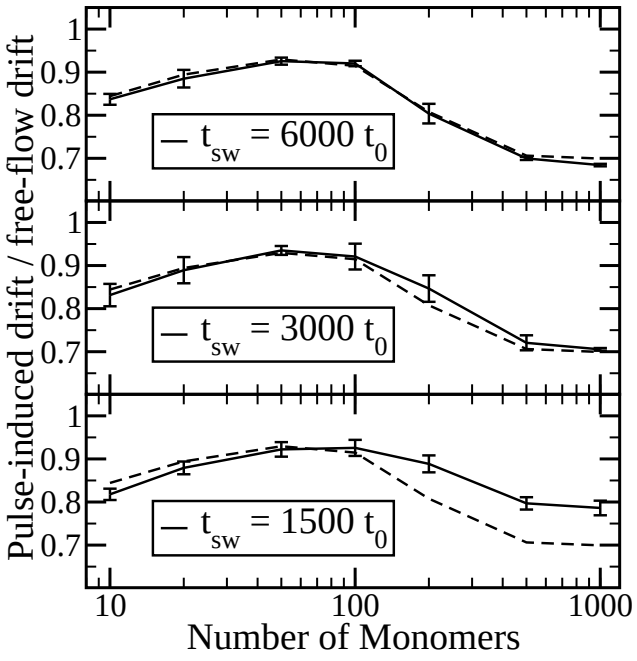

Figure 5.5: Drift lengths as a function of the chain length at $E_1 = 0.2E_0$, for switching times $t_{\mathrm{sw}} = 1500$, 3000, and $6000t_0$. For a better comparison, the drift lengths have been normalized by the free-flow expectation value. The expectation value in the adiabatic limit is shown by the dashed lines.

before, the drift lengths have been normalized with the free-flow expectation value, $\Delta_{x,\mathrm{free}} = 2.5L = 300\sigma$, and the expectation value from the steady-state migration (Ch. 4.5) is shown by a dashed line. Apparently, the drift lengths of all three chains show deviations from the steady-state prediction at high fields. Notable deviations are found at $E_1 \geq 0.5E_0$ for $N = 100$, $E_1 \geq 0.1E_0$ for $N = 200$, and $E_1 \geq 0.05E_0$ for $N = 500$ monomers. A comparison with figure 4.10 yields that in all three cases, the slow state is populated with at least $\sim 20\%$. Thus this analysis underlines the finding presented above. In the fast state, the chains are unaffected by a sudden reversal of the electric field, as the migration conformation does not exhibit an orientation. In the slow state, the migration conformation does exhibit an orientation, which needs to be adjusted after the reversal of the electric field. The time of adaptation τ_{a} can be roughly estimated by the time it takes the polymer to migrate one device length in free flow: $\tau_{\mathrm{a}} \sim L/E_1\mu_0$.

5.4 Asymmetric pulsed electric Field

As clearly indicated by the simulation data presented above, even time-symmetric pulsed electric fields may affect the migration notably. This is caused by a reorientation of the migration orientation in the slow state. However, the application of an electric field with

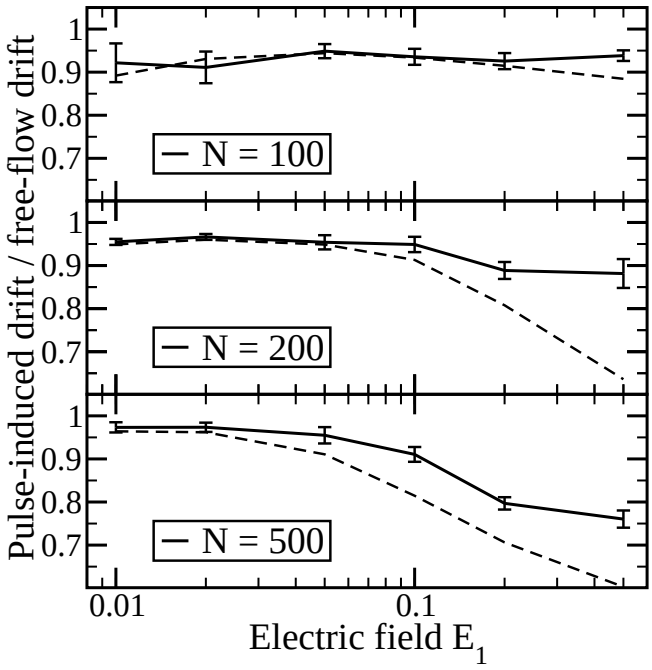

Figure 5.6: Drift lengths as a function of the electric field E_1 for chain lengths $N = 100$, 200, and 500 monomers at switching time $t_{\mathrm{sw}} = 2.5L/E_1\mu_0$. The values have been normalized by the free-flow expectation value. The expectation value in the adiabatic limit is shown by the dashed lines.

a broken time symmetry demands an adaptation to different migration states, which may exhibit very long relaxation times.

5.4.1 Trajectories from Simulation

Trajectories of $N = 10$, 100, and 1000 monomers in an electric field with $t_1 = 1200t_0$, $E_1 = 0.5E_0$, and $t_1/T = 0.2$ are shown in figure 5.7. Note that for a better recognizability, only the position of the center of mass at the field reversal at the end of the strong, forward electric pulse, which is directed in positive x-direction, are shown.

The chain with $N = 10$ monomers seems to migrate slightly in the direction of the strong pulse. Qualitatively, this behavior can be explained by the adiabatic mobilities (Ch 4 and Fig 4.10), as the mobility steadily increases with the field strength and the chain is able to adapt to a new field very quickly. This finding also complies with the probability of getting caught in the wide region (Ch. 3.4.5).

For $N = 100$ monomers, the chain, on average, mostly keeps its position, and almost no drift can be seen. As seen in chapter 5.3.3, these chains are able to adapt to the steady-state value during the pulse.

The situation is different for $N = 1000$ monomers. Here, the chain is not able to adapt to an electric field in any of the pulses. However, a migration in direction of the

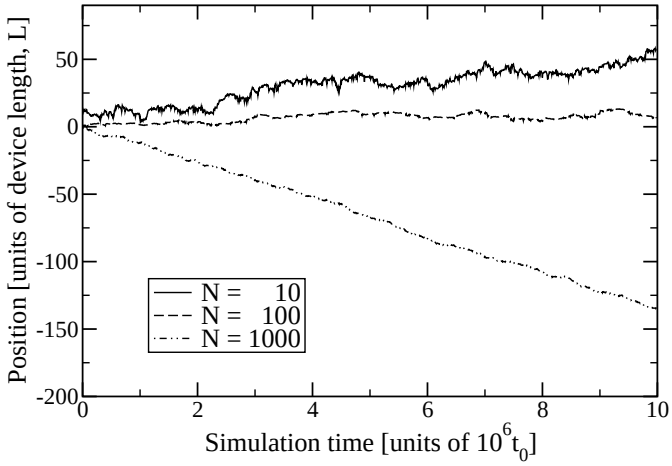

Figure 5.7: Trajectories of $N = 10$, 100, and 1000 monomers in the asymmetric electric field with $t_1 = 1200t_0$, $E_1 = 0.5E_0$, and $t_1/T = 0.2$. For a better recognizability, only the positions at the point of field reversal at the end the strong electric pulse, which is directed in positive x-direction, are shown.

weak pulse nevertheless occurs. To achieve a better understanding of the two migration states during the different pulses, the average z-position of the center of mass has been recorded as a function of the x-position in the unit cell of the device. The result is shown in figure 5.8. The narrow region is marked by a grey background. Note that the center of mass may be located outside the device walls, if the chain is bent around a corner. The figure clearly shows that the chain is pulled into the wide region by the electric field. Furthermore, the penetration depth of the wide region clearly depends on the electric field, and the strong electric pulse pulls the chain further into the wide region than the weak pulse. As the migration speed is directly related to the penetration depth of that region (Fig. 4.5), a net drift in the direction of the weak field occurs. Thus the chain exhibits two different states of migration, even if the fields do not allow for a relaxation into the steady-state migration.

5.4.2 Drift Lengths from Simulation

Drift lengths of $N = 200$ monomers at $E_1 = 0.5E_0$, $t_1 = 5L/E_1\mu_0 = 1200t_0$, and $t_1/T = 0.2$ are shown in figure 5.9. On the right side, the drift length probability densities are shown. Apparently, both drift length histograms exhibit two distinct peaks, as in figure 5.4. The histograms of the forward and backward drift are obviously highly correlated, and they are almost equal. This is also reflected by the drift length probability densities shown in the inset. Thus two states of migration still exist. As a consequence, the drift largely unaffected by the current electric field, and is mostly determined by the previous migration state.

Drift lengths a a function of the chain length in a electric field with $E_1 = 0.5E_0$,

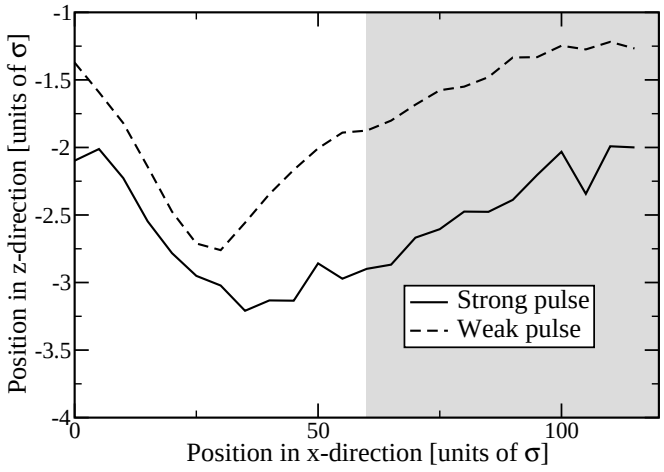

Figure 5.8: Average position of the center of mass for $N = 1000$ monomers during the strong (forward) and weak (backward) pulses inside the device. The narrow region is marked by a grey background. Note that the center of mass may be located outside the device walls, if the chain is bend around a corner.

$t_1 = 1200t_0$, and $t_1/T = 0.2$ are shown in the upper panel of figure 5.10. For comparison, the expected drift lengths in the case of symmetric pulses are given in the middle panel, and the case of a static electric field is shown in the lower panel. For comparison, all values have been normalized by the free flow value $\Delta_{x,\mathrm{free}} = E_1 t_1 = 600\sigma$.

A was already seen in chapter 5.3.3, the steady-state and the symmetric pulse value agree quite well for short chains, and slight deviations occur for long chains. However, these data obviously cannot be applied to an electric field with a broken time symmetry. In this case, the forward drift induced by the strong field and the backward pulse induced by the weak field are almost identical, regardless of the chain length. Slight deviations of these drifts are only found for short chains, where the chains favor the direction of the strong pulse, and for long chains, which favor the direction of the weak pulse.

5.4.3 Migration speeds from Simulation

As the forward and backward drift lengths do not differ significantly in most cases (Fig. 5.10), and are even highly correlated (Fig. 5.3), the analysis of a single drift length is inappropriate. However, slight deviations of the forward and backward drift lengths do exist for long and for short chains. Thus the total drift length

$$\Delta_{x,\mathrm{tot}} = \Delta_{x,\mathrm{forw}} + \Delta_{x,\mathrm{backw}} \tag{5.4}$$

obviously depends on the electric field, and the chain length (Fig. 5.7). Furthermore, this observable offers the advantage that the sum of the two drift lengths do not scatter as much as the single drift lengths. This observable can be easily related to the total

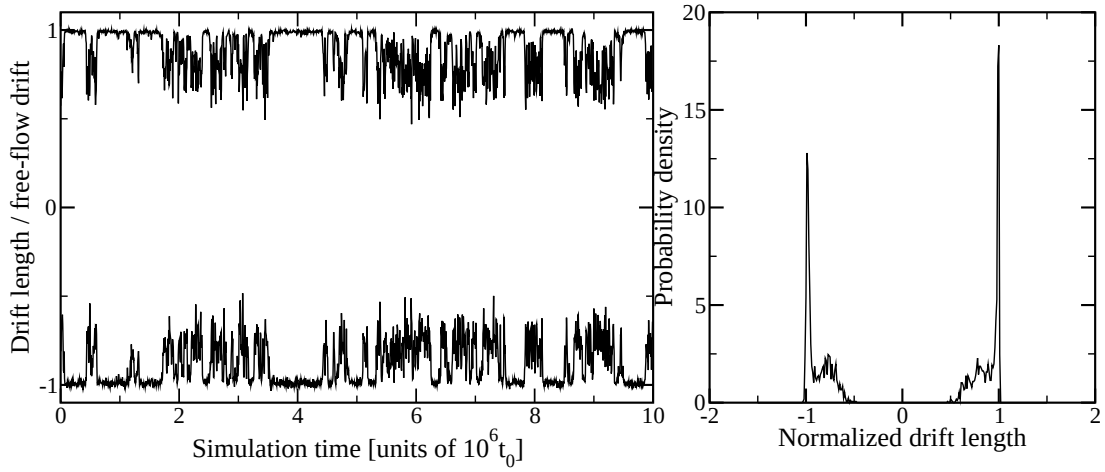

Figure 5.9: Normalized drift lengths of $N = 200$ monomers at $E_1 = 0.5E_0$, $t_1 = 5L/E_1\mu_0 = 1200t_0$, and $t_1/T = 0.2$ (left side). The plot on the right shows drift length histograms of the forward and backward pulses.

drift velocity $\langle v_\mathrm{d} \rangle$:

$$\langle v_\mathrm{d} \rangle = \frac{\langle \Delta_{x,\mathrm{tot}} \rangle}{T} \tag{5.5}$$

Mean migration speeds $\langle v_\mathrm{d} \rangle$ as a function of the chain length in an electric field with $E_1 = 0.2$ and $0.5E_0$, $t_1 = 5L/E_1\mu_0$, and $t_1/T = 0.2$ are shown in figure 5.11. For comparison, the expectation values from static electric fields are also shown. Apparently, the migration speeds in the pulsed field are much lower than expected from the values of static electric field. At $E_1 = 0.2E_0$, the drift velocity $\langle v_\mathrm{d} \rangle$ is very small, and a significant drift could not be detected for any chain length. At $E_1 = 0.5E_0$, however, short chains drift in the direction of the strong pulse, and long chains show a mean drift in the direction of the weak pulse, as already seen in figure 5.7. Thus the electric field E_1 needs to exceed a minimum value for a ratchet effect to occur. This agrees with the idea that the ratchet effect is based on the two-state migration, as this effect also needs a minimum electric field.

5.4.4 Minimum Pulse Time

To find out the minimum switching time, simulations with $t_1 = 1$, 2, 3, and $5L/E_1\mu_0$ at an electric field $E_1 = 0.5E_0$ were performed. The mean drift speeds $\langle v_\mathrm{d} \rangle$ for $t_1 = 1$, 3, and $5L/E_1\mu_0$ are shown in figure 5.12.

At $t_1 = 1$ and $3L/E_1\mu_0$ (240 and $720t_0$), the mean drift velocity $\langle v_\mathrm{d} \rangle$ does not differ significantly from zero. Here, the chains are obviously unable to adapt to two different migration states. The situation is different at $t_1 = 5L/E_1\mu_0 = 1200t_0$. After such an adaptation time, the chains are able to adjust to two different states of migration. With the data shown here, longer pulse times t_1 should result in a greater drift velocity $\langle v_\mathrm{d} \rangle$.

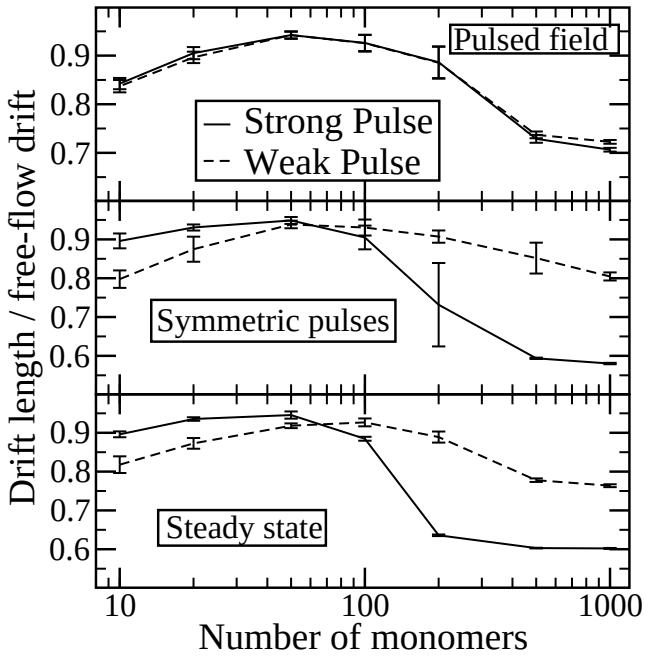

Figure 5.10: Normalized drift lengths as a function of the chain length in the asymmetric electric field with $t_1 = 5L/E_0\mu_0$, $E_1 = 0.5E_0$, and $t_1/T = 0.2$. For comparison, the expectation value for the symmetric electric field is shown in the middle, and the steady-state values are given in the lower panel.

5.4.5 Ratio of the Pulse Times

As already mentioned in chapter 5.2, simulations were carried out at $t_1/T = 0.3$ as well. As described above, only one parameter set exhibits a mean drift of the chains induced by a ratchet effect: $E_1 = 0.5E_0$ and $t_1 = 5L/E_1\mu_0 = 1200t_0$. Mean drift velocities $\langle v_\mathrm{d} \rangle$ obtained from simulation are shown in figure 5.13. Short chains seem to drift in the direction of the strong pulse, and long chains drift in the direction of the weak pulse. Except for a slight difference of the drift velocity for short chains, no notable difference to the data for $t_1/T = 0.2$ is found. One the one hand, one expects the ratchet mechanism to vanish in the limit $t_1/T \to 1/2$ ($t_1 \to t_2$). On the other hand, the two different migration states do not differ that much, and the chains are able to adjust to the new electric field quicker. Here, these two effects seem to cancel each other.

5.4.6 Theoretical plate number

As already mentioned in chapter 3.4.3, the separation efficiency of a device is often measured in terms of the theoretical plate number, N_plate (Eq. 3.5). A net drift is clearly exhibited at $E_1 = 0.5E_0$, $t_1 = 5L/E_0\mu_0$, and $t_1/T = 0.2$ and 0.3 (Fig. 5.13) for long and short chains. Therefore, I have investigated the plate number of these runs.

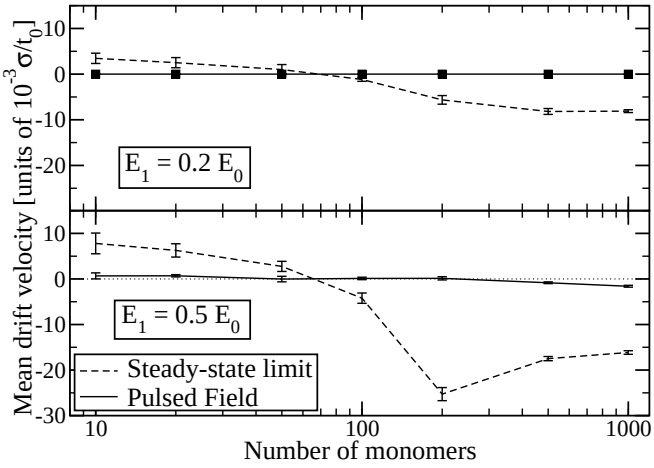

Figure 5.11: Migration speeds as a function of the chain length at $E_1 = 0.2$ and $0.5E_0$, $t_1 = 5L/E_1\mu_0$, and $t_1/T = 0.2$. For comparison, the expectation value from the steady-state limit is also shown by the dotted lines.

These results are shown in table 5.1. Theoretical plate numbers per device roughly cover the range from 2 to 20, which is smaller than for entropic traps. Using a total device length of 10μm, one finds $0.2 - 2 \cdot 10^6$ plates per meter.

Note that in the case of $N = 200$ monomers at $t_1/T = 0.3$, a plate number of the forward drift could be assigned. In this case, the adiabatic limit predicts a backward motion of $(-23 \pm 4) \cdot 10^{-3}\sigma/t_0$. Here, the drift velocity in the pulsed field is $\langle v_\mathrm{d} \rangle = (3.2 \pm 3.4) \cdot 10^{-4}\sigma/t_0$. Thus the chain almost exhibits a significant drift in the opposite direction than the adiabatic prediction.

N	direction	N_plate at $t_1/T = 0.2$	N_plate at $t_1/T = 0.3$
10	forward	3.1	1.8
20	forward	4.8	1.8
50	n.a.	n.a.	n.a.
100	n.a.	n.a.	n.a.
200	forward	n.a.	3.0
500	backward	9.3	8.2
1000	backward	26.2	19.2

Table 5.1: Theoretical plate number N_plate per device for different chain lengths at $E_1 = 0.5E_0$, $t_1 = 5L/E_0\mu_0$, and $t_1/T = 0.2$ and 0.3. Sets with less than 30 samples have not been considered.

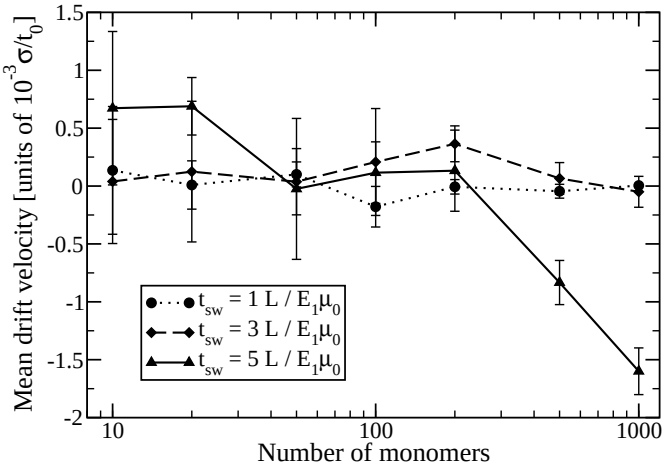

Figure 5.12: Migration speeds as a function of the chain length at $E_1 = 0.5E_0$, $t_1 = \{1, 3, 5\}L/E_1\mu_0$, and $t_1/T = 0.2$.

5.5 Conclusions

To summarize, I have performed an off-lattice Brownian dynamics simulation of a coarse-grained model of DNA in a rectangular array of obstacles with a pulsed electric field with mean zero. The electric field distribution inside the device geometry corresponds to that presented in chapter 4, and thus allows for two distinct migration states. The only difference is that the microchannel is now replaced by an array, see figure 2.7.

In a first stage, I have investigated the effects caused by a time-symmetric electric field, *i. e.* $E_1 = -E_2$, with mean zero. Again, two distinct migration states are found, which exhibit different properties even in the pulsed electric field. For long pulse times, and short chains, the drift widths can be described by the data obtained in the static electric field case. Notable differences occur when the field is strong, and long chains are investigated. These deviations can be explained by a non-vanishing population density of the slow state, which has a migration orientation. The point of divergence of the data has been examined, and a parameter to predict this point from steady-state simulation data has been found.

In a second stage, I have investigated the effects of an electric field with a broken time symmetry, *i. e.* $E_1 \neq -E_2$, with mean zero. Two distinct states of migration still exist, and the migration during a single pulse is almost defined by the previous state of migration. However, I have found that it is possible to exploit the two-state behavior as a ratchet mechanism with moderate switching rates of the electric field. The influence of the electric field strength itself was discussed, and the ratchet mechanism also needs a minimum electric field to occur. I have also investigated the influence of the ratio of the applied electric fields, and did not find a notable difference. This is probably accidental and related to the fact that my set of simulation parameters only covered one set which exhibited the ratchet effect.

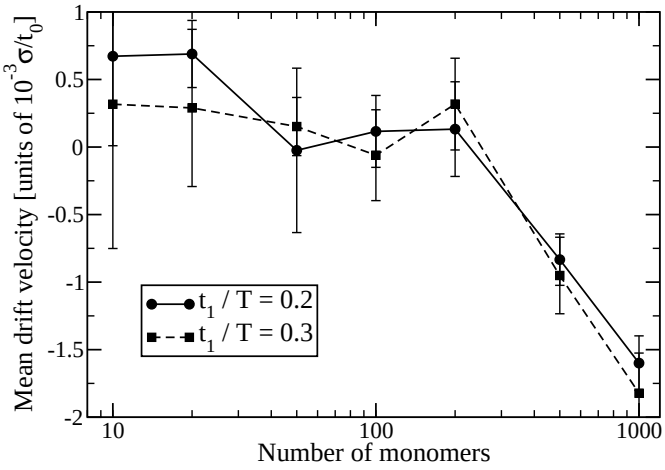

Figure 5.13: Migration speeds as a function of the chain length at $E_1 = 0.5E_0$, $t_1 = 5L/E_1\mu_0$, and $t_1/T = \{0.2, 0.3\}$.

6 Conclusions and Outlook

To summarize, I have presented an off-lattice simulation of a simple bead-spring model, which represents a coarse-grained model of DNA, with Brownian dynamics. In contrast to the preliminary work in my diploma thesis, the electric field in the device is no longer assumed to be homogeneous, and much better agreement with experimental data is achieved. Nevertheless, some important interactions are neglected by the model. It is shown that none of these should affect the simulation data notably, if the parameters of the model are matched to those of the experiment appropriately.

First I have considered entropic traps. My data reproduces the migration order reported by Han and Craighead [53, 54, 55, 56], who have reported that long chains migrate faster than short ones. Furthermore, Han and Craighead have proposed entropic trapping as a length-dependent trapping mechanism. In that case, the escape rate should be proportional to $N^{3/5}$, with N the number of monomers. Indeed, my simulation data supports this picture. However, they also reveal an additional trapping mechanism with an escape rate proportional to $N^{1/5}$. In this case, chains deviate from the main path into the deep region of the device, and get stuck in the field-free region. From my simulation data, an estimate of the mobility due to this new trapping mechanism in the device is obtained. This is compared to the simulation data, and good agreement is found.

I have also investigated the migration of model DNA in geometrically structured microchannels presented by Duong *et al.* [62]. With my data, it is possible to explain the reversal of the chain length dependent mobility reported. This reversal turns out to be the signature of a high-field, non-equilibrium bistability. This bistability has also been observed in experiments as well [63]. A detailed analysis, including the mobilities and the population densities of the two states, is given. At high fields, the main contribution to the chain length dependent mobility comes from the different population densities of the two states. Furthermore, I have shown that this behavior can be found in an array of microchannels as well.

As the two migration states exhibit two distinct migration speeds, which differ by a factor of almost 2, I have also investigated the influence of a pulsed electric field in an array of microchannels. The investigation is performed in two steps: first, the influence of a time-symmetric electric field with mean zero is analyzed. Differences to the steady-state behavior are only found if the slow state is populated notably. As the migration conformations during the two pulses are equal, this underlines that the slow state exhibits an orientation, which does not exist in the fast state. In a second step, I applied an electric field with mean zero, but a broken time symmetry, *i. e.* the field consists of a strong and a weak pulse. My simulation data shows that the two-state migration can indeed be exploited as a ratchet mechanism, given a strong electric field, and if the electric field pulses are applied for a sufficient amount of time. However, the

simulation parameter range did not cover the full range of the ratchet effect, and further investigations are necessary.

As shown above, my simple model captures the essential physics in geometric structured devices like entropic traps and microchannels. As my model incorporates inertia, it is not useful at very high fields, which are necessary to exploit the ratchet effect. On the other hand, my model is inefficient in the overdamped limit. Currently, efforts are being made to use stiff bonds, which hopefully allow for a larger time step in the overdamped case. Another promising way is to use optimized channel geometries, which still need to be developed.

A Modifications of the Simulation Program

The simulation program has already been described in ref. [86]. However, a few extensions are applied. These shall be described below.

A.1 Walls in y-direction

These are necessary to model the finite depth of the device presented in chapter 4. The same potential as that describing the interaction with the other walls is used.

A.2 Inhomogeneous electrical field

A numerical solution for Φ is obtained from the software program "Matlab" (Mathworks, US, `http://www.mathworks.com`). The solution is computed by a finite element solver with an adaptive triangulation and evaluated on a square grid. The grid values are read in by the simulation program. To compute the electric field at a given point $\vec{r}$, the grid values of the derivatives are interpolated bilinearly.

A.2.1 Potential File

The solution $\Phi(x, z)$ on a square lattice has to be stored in a file that needs to fulfill the following requirements. Basically, this file consists of two parts: a header line, and the raw data.

An example header line looks like this:

```
# 1 60 60 60 60 0 0.5 0.0083333 58081
```

Parts of that line are used by the simulation program to determine whether the file fits to the parameter set described in the `constants` file (App. B). The meaning of the device size parameters is described in chapter 2.8. From left to right, the meaning of the entries is interpreted by the simulation program to be:

- #: To comment out the header line when importing the data into another program, like `XmGrace`.
- 1: The version of the file, 1 representing the microchannel setup described in chapter 2.8. Up to now, this is the only version of file supported.
- 60: The height h_t of the narrow region.
- 60: The height h_w of the wide region.

60: The length l_w of the wide region.

60: The length L_t of the narrow region.

0: The tilt of the walls.

0.5: Represents the distance of the grid points, h. As the simulation program needs a square grid, this represents the distance of the grid points in both the x and the z direction.

0.0083333: This entry characterizes the electric field strength of the potential Φ. In this case, $\Phi(90, z) - \Phi(-30, z) = 1$ is fulfilled, resulting in a mean field strength $E = 1/120 E_0$.

58081: The number of data sets stored in the file. Here, the unit cell of the device has $120 \times 120\sigma^2$. With the grid distance $h = 0.5\sigma$, this leads to a grid of $241 \times 241 = 58081$ data sets.

The data sets in the file consist of three values: x, z, and $\Phi(x, z)$, and are stored in this order. However, the ranges covered need to fulfill (Fig. A.1):

x **direction:** The range from $-l_\mathrm{t}/2$ up to $l_\mathrm{w} + l_\mathrm{t}/2$

z **direction:** The range from $-h_\mathrm{w}$ up to h_t

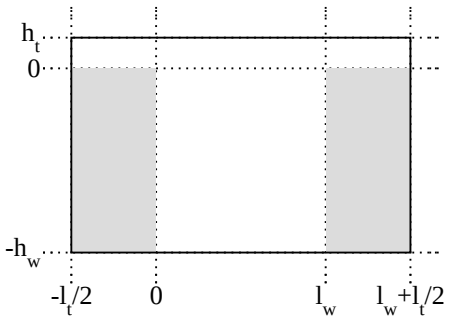

Figure A.1: Ranges covered by the potential file. All points need to be inside the area shown, with the coordinate system indicated. Inside the walls (gray area), data points may be given, as the simulation program neglects these.

These ranges need to be fully covered by the square grid. Inside the walls (gray areas), data sets may be given as well, as these are neglected during evaluation. Note that the simulation program is not able to handle the string "NaN" (Not a Number) properly. This is given as a result by the software program "Matlab" when computing the solution outside the device boundaries.

A.3 Observables

For a simpler analysis, some observables were added to the simulation program, which originally only recorded the center of mass and the positions of the most extended monomers. These are:

- First Rouse mode $X_1 = \frac{1}{N}\sum_{i=1}^{N} \cos\left(\frac{(n+\frac{1}{2})\pi}{N}\right) \vec{r}_n$: This observable allows for an analysis of the conformational relaxation time, as it is the Rouse mode which has the longest decorrelation time.
- Radius of Gyration $R_{g,\alpha}^2 = \frac{1}{N}\sum_n (r_{n,\alpha} - R_{G,\alpha})^2$ for each configuration and each direction $\alpha = \{x, y, z\}$, where R_G is the center of mass
- End-to-End vector $\vec{R} = \vec{r}_N - \vec{r}_1$
- The measurement of the most extended monomers now records both monomer index i and full position $\vec{r}_i$
- The contour length $c = \sum_{n=1}^{N-1} |\vec{r}_{n+1} - \vec{r}_n|$
- The velocity of the center of mass $\vec{v}_G = \frac{1}{N}\sum_n \vec{v}_n$
- The angular momentum in the center of mass system $\vec{L} = \sum_i (\vec{r}_i - \vec{R}_G) \otimes \vec{v}_i$
- The energy of the polymer, split into purely kinetic energy of the monomers, both external (walls) and internal (Lennard-Jones, harmonic spring) interaction energy
- The mean cosine of the bond angle, $< \cos\Theta >$
- The scalar product of the first bond with the end-to-end vector, $(\vec{r}_2 - \vec{r}_1) \cdot (\vec{r}_N - \vec{r}_1)$
- The longest and the shortest bond length in a given snapshot. This allows for ensuring that no bond crossing occurs during simulation (Ch. 2.6, ref. [86]).
- The current multiplication factor τ for the electric field (Ch. 2.9.1). This is for checking the correct implementation of the code as well as analysis.

Furthermore, the simulation program is now capable of appending configuration snapshots regularly to a file, allowing for a detailed analysis afterwards.

A.4 Computation of the Lennard-Jones Potential

The data structure `hash_map` [111] is no longer supported by the Standard Template Library (STL). Therefore, the simpler and slightly slower data structure `map` is used, which is based on a red-black search tree. However, recent advances in compiler technology ensure a $\approx 40\%$ gain in speed of the compiled code.

A.5 Overdamped Dynamics

For special tests, dynamics in the limit $m \to 0$ have been implemented. A detailed description is given in chapter 2.1.2 and 2.2.2.

A.6 Pulsed Field

As shown in chapter 2.9.1, four parameters are needed to fully describe any rectangular pulsed electric field. In the simulation program, the parameters E_1, $\tau = E_2/E_1$, Δt_1 and $T = \Delta t_1 + \Delta t_2$ have been implemented. It is checked which of the two fields has to be applied in each time step. For a simpler implementation, the electric force is computed for $E = E_1$ in every case, and multiplied with a factor of either one or τ afterwards.

A.7 Parallelization of the Device

To compute the forces inside the array device (fig. 2.7), the particle position is transformed into the unit cell shown. If necessary, it is mirrored into the unit cell of the simple device (fig. 2.6). Then the forces are computed and, if necessary, retransformed.

B Source Code of the Simulation Program

Below you find the complete sources of the simulation program, a sample parameter file (`constants`), a sample random seed file (`random-seed`), and a list of possible error codes and the meaning thereof
(`errorlist.txt`).

The initial configuration file contains the time steps already computed in the first line, and the number of monomers stored in the file in the second. In the next lines, the locations and velocities of the monomers are given, in the x-, y-, and z-direction. A sample configuration file of $N = 10$ monomers after 0 time steps looks like:

```
# 0
# 10
40 0 -40  0 0 0
40 1 -40  0 0 0
40 2 -40  0 0 0
40 3 -40  0 0 0
40 4 -40  0 0 0
40 5 -40  0 0 0
40 6 -40  0 0 0
40 7 -40  0 0 0
40 8 -40  0 0 0
40 9 -40  0 0 0
```

config_utils.cc

```
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include"point.hh"
#include"constants.hh"

#ifndef CONFIG_UTILS_INCLUDED
#define CONFIG_UTILS_INCLUDED

// This file contains a few routines to handle configuration snapshots, and to perform a few
// basic measurements.
// Functions used:
// - rouse_mode       : computes the rouse-modes of a configuration
// - delta            : simple Kronecker-delta
// - theta            : mean of the cosine of the bond-angles
// - gyration-radius  : computes the raius of gyration of a configuration
// - angular momentum : computes the angular momentum of a configuration
// - valid_checkpoint : checks the validity of a checkpoint file
// - read_config      : reads in a configuration from a file
// - write_config     : writes a configuration to a file
// -count_config      : counts the number of snapshots in a configuration file

using namespace std;

const int number_of_blocks = 10;

point rouse_mode(point* r, int N, int mode) {
  point help = point(0,0,0);
  for (int i = 0; i < N; i++)
    help += cos(mode * M_PI * (i+0.5) / N) * r[i];
  help /= (N);
  return help;
}

int delta(int i, int j) {
  return (i==j) ? 1 : 0;
}

double theta(point* r, int number){
  double sum_theta = 0;
  for (int i = 1; i<number-1; i++)
    sum_theta += (r[i-1]-r[i])*(r[i]-r[i+1])/(abs(r[i-1]-r[i])*abs(r[i]-r[i+1]));
  if (number > 2)
    return sum_theta/(number-2);
  else
    return 0;
}

point gyration_radius(point* r, int N){
  point center = point(0,0,0);
  for (int i = 0; i < N; i++)
    center += r[i];
  center /= N;
  double sum_sqr_x = 0;
  double sum_sqr_y = 0;
  double sum_sqr_z = 0;
  for (int i = 0; i < N; i++) {
    sum_sqr_x += (r[i] - center).x_() * (r[i] - center).x_();
    sum_sqr_y += (r[i] - center).y_() * (r[i] - center).y_();
    sum_sqr_z += (r[i] - center).z_() * (r[i] - center).z_();
  }
  return point(sqrt(sum_sqr_x/N), sqrt(sum_sqr_y/N), sqrt(sum_sqr_z/N));
}

point angular_momentum(point* r, point* v, int number) {
  point center = rouse_mode(r, number, 0);
  point sum_ang_momentum = point(0,0,0);
  for (int i = 0; i < number; i++)
    sum_ang_momentum += vectorprod(r[i]-center, v[i]);
  return sum_ang_momentum;
}

int valid_checkpoint(char* start_name, char* checkpoint_name, int number) {
  char* command = new char[2*strlen(start_name) + 2*strlen(checkpoint_name) + 100];
  sprintf(command, "rm -f %s.tmp", checkpoint_name);
  system(command);                              // Avoid race condition when starting multiple jobs simultaneously
  sprintf(command, "head -n %i %s | wc -l > %s.tmp ; head -n %i %s | grep -v nan | wc -w >> %s.tmp",
          number + 2, start_name, checkpoint_name, number + 2, start_name, checkpoint_name);
  if (system(command)) {                        // Check whether file exists
    sprintf(command, "rm -f %s.tmp", checkpoint_name);
    system(command);                            // Remove temporary file
    return 0;                                   // Invalid start-file (not found)
  }
  sprintf(command, "%s.tmp", checkpoint_name); // command is used as filename in this case
  FILE* input = fopen(command, "r");
  int lines, words;
  fscanf(input, "%i %i", &lines, &words);
  sprintf(command, "rm -f %s.tmp", checkpoint_name);
  system(command);                              // Remove temporary file
  if ((lines != number+2) || (words != 6*number + 4)) { // Check number of lines and words
    return 0;                                   // Invalid start-file (incomplete)
  }
  return 1;                                     // No errors detected
}

void read_config(FILE*& input, point* r, point* v, int number, int& timesteps, double& energy,
                 int init = 0) {
  int number2;
  fscanf(input, "%*s %i %*s %i", &timesteps, &number2);
  for (int i = 0; i < number; i++) {
    double x, y, z, vx, vy, vz;
    fscanf(input,"%lf %lf %lf %lf %lf %lf", &x, &y, &z, &vx, &vy, &vz);
    r[i] = point( x,  y,  z);
```

config_utils.cc

```
    v[i] = point(vx, vy, vz);
  }
  if (!init)                                  // Start-files need not have a valid energy
    fscanf(input, "%*s %lf" ,&energy);
}

void write_config(point* r, point* v, double energy, int number, int total_timesteps,
                  char* filename, char* open_mode) {
  FILE* out;
  out = fopen(filename, open_mode);
  fprintf(out, "# %i\n", total_timesteps);
  fprintf(out, "# %i\n", number);
  for (int k = 0; k<number; k++) {
    fprintf(out, "%lf %lf %lf  ", r[k].x_(), r[k].y_(), r[k].z_());
    fprintf(out, "%lf %lf %lf \n", v[k].x_(), v[k].y_(), v[k].z_());
  }
  fprintf(out, "# %lf\n", energy);
  fprintf(out, "\n");
  fclose(out);
}

int count_configs(char* filename) {
  int number_of_monomers;
  FILE *input = fopen(filename, "r");
  fscanf(input, "%*s %*i %*s %i", &number_of_monomers); // Determine the number of
  fclose(input);                                        // monomers in the file
  int number_of_configs;
  char *command = new char[100 + strlen(filename)];
  system("rm -f tmp.conf");
  sprintf(command, "wc %s > tmp.conf", filename);  // Read the number of lines
  system(command);                                 // in the file
  input = fopen("tmp.conf", "r");
  fscanf(input, "%i", &number_of_configs);
  number_of_configs /= (number_of_monomers+4);     // and compute the number of configs
  fclose(input);
  system("rm -f tmp.conf");
  return number_of_configs;
}

#endif
```

config_utils.hh

```
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
#include"point.hh"
#include"constants.hh"

#ifndef CONFIG_UTILS_INCLUDED
#define CONFIG_UTILS_INCLUDED

// This file contains a few routines to handle configuration snapshots, and to perform a few
// basic measurements.
// Functions used:
// - rouse_mode       : computes the rouse-modes of a configuration
// - delta            : simple Kronecker-delta
// - theta            : mean of the cosine of the bond-angles
// - gyration-radius  : computes the raius of gyration of a configuration
// - angular momentum : computes the angular momentum of a configuration
// - valid_checkpoint : checks the validity of a checkpoint file
// - read_config      : reads in a configuration from a file
// - write_config     : writes a configuration to a file
// -count_config      : counts the number of snapshots in a configuration file

using namespace std;

const int number_of_blocks = 10;

point rouse_mode(point* r, int N, int mode);

int delta(int i, int j);

double theta(point* r, int number);

point gyration_radius(point* r, int number);

point angular_momentum(point* r, point* v, int number);

int valid_checkpoint(char* filename, char* checkpoint_name, int number);

void read_config(FILE*& input, point* r, point* v, int number, int& timesteps, double& energy,
                 int init = 0);

void write_config(point* r, point* v, double energy, int number, int total_timesteps,
                  char* filename, char* open_mode);

int count_configs(char* filename);

#endif
```

constants

```
version                 7
use_verlet              1
enable_LJ               1
backbone_LJ             1
enable_BL               0
enable_BA               0
enable_FR               1
enable_BM               1
enable_SP               1
enable_trap             1
enable_mirror_trap      0
enable_wall_y           1
enable_el               0
enable_el_trap          1
enable_pulse_el         0
display_mode_1          1
display_gyration        1
display_end_to_end      1
display_min_max         0
display_min_max_mon     0
display_min_max_all     1
display_contour         1
display_min_max_bond    0
display_mean_speed      1
display_ang_momentum    1
display_energy          1
display_split_energy    1
display_cos_theta       1
display_bond_correl     1
display_current_factor  0
display_constants       1

Number_of_steps         1000
Output_steps            100
Config_steps            100
throw_away              0
Time-step               0.010000

zeta_FR                 1.000000

Temperature             1.000000
epsilon_BM              1.000000
save_new_seed           1

epsilon_SP              100.000000

epsilon_LJ              1.000000
sigma_LJ                1.000000

epsilon_BL              10.000000
d_BL                    1.000000
d_0_BL                  1.000000

epsilon_BA              5.000000

epsilon_LJ_wall         1.000000
sigma_LJ_wall           1.000000
l_cutoff_LJ_wall        1.122462

h_top_trap              60.000000
h_bottom_trap           60.000000
l_bottom_trap           60.000000
l_top_trap              60.000000
wall_tilt_trap          0.000000

wall_y                  60.000000

epsilon_el_x            -0.010000
epsilon_el_y            0.000000
epsilon_el_z            0.000000

epsilon_el_trap         -0.040000
potential_file_trap     /scratch_cm/streek/real_field_60_60_60_60/misc/field

pulse_period_el         0
pulse_up_el             0
pulse_factor_el         0
```

constants.cc

```
#include<iostream>
#include<fstream>
#include<string>
#include<math.h>
#include<stdio.h>

#ifndef CONSTANTS_INCLUDED
#define CONSTANTS_INCLUDED

using namespace std;

// This file reads the constants for the main program and a function for displaying
// them, if desired.

// These are the switches for the potentials, algorithms and output
int use_verlet;
int enable_LJ;
int backbone_LJ;
int enable_BL;
int enable_BA;
int enable_FR;
int enable_BM;
int enable_SP;
int enable_trap;
int enable_mirror_trap;
int enable_wall_y;
int enable_el;
int enable_el_trap;
int enable_pulse_el;
int display_constants;
int display_min_max;
int display_min_max_mon;
int display_min_max_all;
int display_contour;
int display_min_max_bond;
int display_gyration;
int display_end_to_end;
int display_mode_1;
int display_mean_speed;
int display_ang_momentum;
int display_energy;
int display_split_energy;
int display_cos_theta;
int display_bond_correl;
int display_current_factor;
int displayed_variables;
int save_new_seed;
int append_configs;

// This is the number of steps
int steps;
// This is the number of displayed configurations
int output_steps;
// This is the number of displayed configurations in full
int config_steps;
// This is the number of thwrow-away-steps for initialisation
int throw_away;

// This is the time-step
double delta_t;
double sqrt_delta_t;

// This is the friction-constant
double zeta_FR;
double decay_v;

// This is the temperature
double T;
double sqrt_T;

// This is for the brownian motion
double epsilon_BM;
double prefactor_BM;
unsigned long rand_seed;

// This is the spring-constant
double epsilon_SP;
double prefactor_SP;

// These are the constants for the Lennard-Jones potential between two monomers
double epsilon_LJ;
double sigma_LJ;
double v_cutoff_LJ;
double l_cutoff_LJ;
double l_cutoff_LJ_sqr;
double sigma_6;
double prefactor_LJ;

// These are the constants for the bond-length-energy between two monomers
// on the chain
double epsilon_BL;
double d_BL;
double d_BL_sqr;
double d_0_BL;
double prefactor_BL;

// This is the constant for the bond-angle-energy
double epsilon_BA;
double prefactor_BA;

//This is the wall Lennard-Jones-potential
double epsilon_LJ_wall;
double sigma_LJ_wall;
```

constants.cc

```
double sigma_6_wall;
double l_cutoff_LJ_wall;
double l_cutoff_LJ_wall_sqr;
double v_cutoff_LJ_wall;
double prefactor_LJ_wall;

//This is the entropic trap/structured microchannel
double h_top_trap;
double h_bottom_trap;
double l_bottom_trap;
double l_top_trap;
double wall_tilt_trap;
double l_surface_trap;
double h_total_trap;

//This is the wall in y-direction for entropic trap
double wall_y;

// This is the homogeneous external electric field
double epsilon_el_x;
double epsilon_el_y;
double epsilon_el_z;
double prefactor_el_x;
double prefactor_el_y;
double prefactor_el_z;

//This is the electrical field inside the trap
double epsilon_el_trap;
char* potential_file_trap;
int trap_max_x = 0;
int trap_max_z = 0;
int trap_top_x = 0;
int trap_top_z = 0;
double trap_scaling_factor = 0;
double trap_discretize = 0;

//This the pulsed electrical field
int pulse_period_el;      // Pulse period(=(uptime+downtime) in timesteps
int pulse_uptime_el;      // Uptime begins at beginning of period
double pulse_factor_el;   // Rescale-factor in the downtime
double current_factor_el; // Changes during simulation between 1 and Rescale-factor

// This is for analyzing the output-files
char** stdout_component;

// ------------------------------------------------------------------------------

int create_output_strings(void){
  int column = 1;
  stdout_component = new char*[displayed_variables+2];
  stdout_component[column    ] = "time-step";
  stdout_component[column + 1] = "center-of-mass.x";
  stdout_component[column + 2] = "center-of-mass.y";
  stdout_component[column + 3] = "center-of-mass.z";
  column += 4;
  if (display_mode_1) {
    stdout_component[column    ] = "rouse-mode-1.x";
    stdout_component[column + 1] = "rouse-mode-1.y";
    stdout_component[column + 2] = "rouse-mode-1.z";
    column += 3;
  }
  if (display_gyration) {
    stdout_component[column    ] = "radius-of-gyration.x";
    stdout_component[column + 1] = "radius-of-gyration.y";
    stdout_component[column + 2] = "radius-of-gyration.z";
    column += 3;
  }
  if (display_end_to_end) {
    stdout_component[column    ] = "end_to_end.x";
    stdout_component[column + 1] = "end_to_end.y";
    stdout_component[column + 2] = "end_to_end.z";
    column += 3;
  }
  if (display_min_max) {
    stdout_component[column    ] = "min-value.x";
    stdout_component[column + 1] = "min-value.y";
    stdout_component[column + 2] = "min-value.z";
    stdout_component[column + 3] = "max-value.x";
    stdout_component[column + 4] = "max-value.y";
    stdout_component[column + 5] = "max-value.z";
    column += 6;
  }
  if (display_min_max_mon) {
    stdout_component[column    ] = "min-monomer.x";
    stdout_component[column + 1] = "min-monomer.y";
    stdout_component[column + 2] = "min-monomer.z";
    stdout_component[column + 3] = "max-monomer.x";
    stdout_component[column + 4] = "max-monomer.y";
    stdout_component[column + 5] = "max-monomer.z";
    column += 6;
  }
  if (display_min_max_all) {
    stdout_component[column    ] = "min_x.mon";
    stdout_component[column +  1] = "min_x.x";
    stdout_component[column +  2] = "min_x.y";
    stdout_component[column +  3] = "min_x.z";
    stdout_component[column +  4] = "min_y.mon";
    stdout_component[column +  5] = "min_y.x";
    stdout_component[column +  6] = "min_y.y";
    stdout_component[column +  7] = "min_y.z";
    stdout_component[column +  8] = "min_z.mon";
    stdout_component[column +  9] = "min_z.x";
    stdout_component[column + 10] = "min_z.y";
```

constants.cc

```
    stdout_component[column + 11] = "min_z.z";
    stdout_component[column + 12] = "max_x.mon";
    stdout_component[column + 13] = "max_x.x";
    stdout_component[column + 14] = "max_x.y";
    stdout_component[column + 15] = "max_x.z";
    stdout_component[column + 16] = "max_y.mon";
    stdout_component[column + 17] = "max_y.x";
    stdout_component[column + 18] = "max_y.y";
    stdout_component[column + 19] = "max_y.z";
    stdout_component[column + 20] = "max_z.mon";
    stdout_component[column + 21] = "max_z.x";
    stdout_component[column + 22] = "max_z.y";
    stdout_component[column + 23] = "max_z.z";
    column += 24;
  }
  if (display_contour) {
    stdout_component[column] = "contour";
    column += 1;
  }
  if (display_min_max_bond) {
    stdout_component[column    ] = "min_bond_length";
    stdout_component[column + 1] = "max_bond_length";
    column += 2;
  }
  if (display_mean_speed) {
    stdout_component[column    ] = "mean_speed.x";
    stdout_component[column + 1] = "mean_speed.y";
    stdout_component[column + 2] = "mean_speed.z";
    column += 3;
  }
  if (display_ang_momentum) {
    stdout_component[column    ] = "ang_momentum.x";
    stdout_component[column + 1] = "ang_momentum.y";
    stdout_component[column + 2] = "ang_momentum.z";
    column += 3;
  }
  if (display_energy) {
    stdout_component[column] = "energy";
   column += 1;
  }
  if (display_split_energy) {
    stdout_component[column    ] = "energy-pot-int";
    stdout_component[column + 1] = "energy-pot-ext";
    stdout_component[column + 2] = "energy-kin";
    column += 3;
  }
  if (display_cos_theta) {
    stdout_component[column] = "cos-theta";
    column += 1;
  }
  if (display_bond_correl) {
    stdout_component[column] = "bond-correl";
    column += 1;
  }
  if (display_current_factor) {
    stdout_component[column] = "current_factor_el";
    column += 1;
  }
}

void init_constants(char* filename) {
  FILE* input = fopen(filename, "r");
  if (!input) {
    cerr << "Invalid constants file-name!\n";
    exit(2);
  }
// Check for correct version of constants-file
  int version;
  fscanf(input, "%*s%i", &version);
  if (version != 7) {
    cerr << "Wrong version of constants-file!\n";
    exit(3);
  }
// Initialisation of the switches
  fscanf(input, "%*s%i", &use_verlet);
  fscanf(input, "%*s%i", &enable_LJ);
  fscanf(input, "%*s%i", &backbone_LJ);
  fscanf(input, "%*s%i", &enable_BL);
  fscanf(input, "%*s%i", &enable_BA);
  fscanf(input, "%*s%i", &enable_FR);
  fscanf(input, "%*s%i", &enable_BM);
  fscanf(input, "%*s%i", &enable_SP);
  fscanf(input, "%*s%i", &enable_trap);
  fscanf(input, "%*s%i", &enable_mirror_trap);
  fscanf(input, "%*s%i", &enable_wall_y);
  fscanf(input, "%*s%i", &enable_el);
  fscanf(input, "%*s%i", &enable_el_trap);
  fscanf(input, "%*s%i", &enable_pulse_el);
  fscanf(input, "%*s%i", &display_mode_1);
  fscanf(input, "%*s%i", &display_gyration);
  fscanf(input, "%*s%i", &display_end_to_end);
  fscanf(input, "%*s%i", &display_min_max);
  fscanf(input, "%*s%i", &display_min_max_mon);
  fscanf(input, "%*s%i", &display_min_max_all);
  fscanf(input, "%*s%i", &display_contour);
  fscanf(input, "%*s%i", &display_min_max_bond);
  fscanf(input, "%*s%i", &display_mean_speed);
  fscanf(input, "%*s%i", &display_ang_momentum);
  fscanf(input, "%*s%i", &display_energy);
  fscanf(input, "%*s%i", &display_split_energy);
  fscanf(input, "%*s%i", &display_cos_theta);
  fscanf(input, "%*s%i", &display_bond_correl);
  fscanf(input, "%*s%i", &display_current_factor);
```

constants.cc

```
  fscanf(input, "%*s%i", &display_constants);
  if (display_min_max_all) {
    display_min_max     = 0;
    display_min_max_mon = 0;
  }
  displayed_variables = 4 + 3*display_mode_1 + 3*display_gyration + 3*display_end_to_end +
                        6*display_min_max + 6*display_min_max_mon + 24*display_min_max_all +
                        display_contour + 2*display_min_max_bond + 3*display_mean_speed +
                        3*display_ang_momentum + display_energy + 3*display_split_energy +
                        display_cos_theta + display_bond_correl + display_current_factor;
//   if ((enable_el_trap && !enable_trap)) {
//     cerr << "Incompatible switches - Field in trap enabled, but trap missing!\n";
//     exit(18);
//   }

// Initialisation of steps + outputs + timestep
  fscanf(input, "%*s%i", &steps);
  fscanf(input, "%*s%i", &output_steps);
  fscanf(input, "%*s%i", &config_steps);
  fscanf(input, "%*s%i", &throw_away);
  fscanf(input, "%*s%lf", &delta_t);
  sqrt_delta_t = sqrt(delta_t);
  if ((config_steps % output_steps != 0) || (steps % output_steps != 0)) {
    cerr << "Incompatible step-sizes!\n";
    exit(11);
  }

// Initialisation of friction
  fscanf(input, "%*s%lf", &zeta_FR);
  if (!use_verlet)
    zeta_FR = 1;
  decay_v = (1-delta_t * zeta_FR);

// Initialisation of temperature + brownian motion
  fscanf(input, "%*s%lf", &T);
  sqrt_T = std::sqrt(T);
  fscanf(input, "%*s%lf", &epsilon_BM);                  // epsilom_BM equals the stephan-boltzmann-constant
  prefactor_BM = std::sqrt(40 * epsilon_BM * T * (1-delta_t*zeta_FR/2) * zeta_FR * delta_t);
  fscanf(input, "%*s%i", &save_new_seed);
  FILE* rseed = fopen("random_seed", "r");
  if (!rseed) {
    cerr << "Random-seed not found!\n";
    exit(4);
  }
  fscanf(rseed, "%lu", &rand_seed);
  fclose(rseed);

// Initialisation of spring
  fscanf(input, "%*s%lf", &epsilon_SP);
  prefactor_SP = epsilon_SP * delta_t / 2;

// Initialisation of Lennard-Jones
  fscanf(input, "%*s%lf", &epsilon_LJ);
  fscanf(input, "%*s%lf", &sigma_LJ);
  v_cutoff_LJ = epsilon_LJ * 0.25;
  l_cutoff_LJ =  sigma_LJ * 1.1224620483093;
  l_cutoff_LJ_sqr = l_cutoff_LJ * l_cutoff_LJ;
  sigma_6 = pow(sigma_LJ, 6);
  prefactor_LJ = epsilon_LJ * sigma_6 * 6 * delta_t / 2;

// Initialisation of bond-length
  fscanf(input, "%*s%lf", &epsilon_BL);
  fscanf(input, "%*s%lf", &d_BL);
  fscanf(input, "%*s%lf", &d_0_BL);
  d_BL_sqr = d_BL * d_BL;
  prefactor_BL = epsilon_BL * delta_t / 2;

// Initialisation of bond-angle
  fscanf(input, "%*s%lf", &epsilon_BA);
  prefactor_BA = epsilon_BA * delta_t / 2;

// Initialisation of wall-Lennard-Jones
  fscanf(input, "%*s%lf", &epsilon_LJ_wall);
  fscanf(input, "%*s%lf", &sigma_LJ_wall);
  fscanf(input, "%*s%lf", &l_cutoff_LJ_wall);
  l_cutoff_LJ_wall *= sigma_LJ_wall;
  sigma_6_wall = pow(sigma_LJ_wall, 6);
  v_cutoff_LJ_wall = epsilon_LJ_wall *
    (pow(sigma_LJ_wall/l_cutoff_LJ_wall, 6) - pow(sigma_LJ_wall/l_cutoff_LJ_wall, 12));
  l_cutoff_LJ_wall_sqr = pow(l_cutoff_LJ_wall, 2);
  prefactor_LJ_wall = epsilon_LJ_wall * sigma_6_wall * 6 * delta_t / 2;

// Initialisation of entropic trap/ structured microchannel
  fscanf(input, "%*s%lf", &h_top_trap);
  fscanf(input, "%*s%lf", &h_bottom_trap);
  fscanf(input, "%*s%lf", &l_bottom_trap);
  fscanf(input, "%*s%lf", &l_top_trap);
  fscanf(input, "%*s%lf", &wall_tilt_trap);
  l_surface_trap = l_bottom_trap + l_top_trap;
  h_total_trap = h_top_trap + h_bottom_trap;
  if (enable_mirror_trap && (wall_tilt_trap != 0)) {
    cerr << "Tilt trap is only allowed non-mirrored!\n";
    exit(19);
  }

// Initialisation of wall in y-direction
  fscanf(input, "%*s%lf", &wall_y);

// Initialisation of homogeneous external electric field
  fscanf(input, "%*s%lf", &epsilon_el_x);
  fscanf(input, "%*s%lf", &epsilon_el_y);
  fscanf(input, "%*s%lf", &epsilon_el_z);
  prefactor_el_x = epsilon_el_x * delta_t / 2;
```

constants.cc

```
  prefactor_el_y = epsilon_el_y * delta_t / 2;
  prefactor_el_z = epsilon_el_z * delta_t / 2;

// Initialisation of electric field inside trap
  {
    fscanf(input, "%*s%lf", &epsilon_el_trap);
    char* temp_name = new char[1000];
    fscanf(input, "%*s %s", temp_name);
    potential_file_trap = new char[strlen(temp_name)+1];
    strcpy(potential_file_trap, temp_name);
    delete [] temp_name;
    if (enable_el_trap) {
      double tmp_trap_l_top;
      double tmp_trap_h_top;
      double tmp_trap_l_bot;
      double tmp_trap_h_bot;
      double tmp_tilt_trap;
      double tmp_trap_discretize;
      double tmp_trap_epsilon;
      int   type;
      FILE* input_potential = fopen(potential_file_trap, "r");
      if (!input_potential) {
        cerr << "Potential-file not found!\n";
        exit(9);
      }
      fscanf(input_potential, "%*s%i%lf%lf%lf%lf%lf%lf", &type, &tmp_trap_h_top, &tmp_trap_h_bot,
             &tmp_trap_l_bot, &tmp_trap_l_top, &tmp_tilt_trap, &tmp_trap_discretize, &tmp_trap_epsilon);
      fclose(input_potential);
      if ((tmp_trap_l_top != l_top_trap) || (tmp_trap_l_bot != l_bottom_trap) ||
          (tmp_trap_h_top != h_top_trap) || (tmp_trap_h_bot != h_bottom_trap) ||
          (tmp_tilt_trap != 0) || (type != 1)) {
        cerr << "Potential-file and/or geometry do not fit!\n";
        exit(10);
      }
      trap_scaling_factor = epsilon_el_trap/tmp_trap_epsilon;
      trap_discretize = tmp_trap_discretize;
      trap_max_x = int(std::floor(l_surface_trap / trap_discretize + 0.5));
      trap_max_z = int(std::floor((h_top_trap+h_bottom_trap) / trap_discretize + 0.5));
      trap_top_x = int(std::floor(l_top_trap / trap_discretize + 0.5));
      trap_top_z = int(std::floor(h_top_trap / trap_discretize + 0.5));
    }
  }

// Read parameters for pulsed electrical field
  fscanf(input, "%*s%i",  &pulse_period_el);
  fscanf(input, "%*s%i",  &pulse_uptime_el);
  fscanf(input, "%*s%lf", &pulse_factor_el);
  current_factor_el = 1;
  // current_factor_el will be properly set after reading checkpoint/ startfile

// Correct prefactors if necessary
  if (!use_verlet) {
    prefactor_SP      *=2;
    prefactor_LJ      *=2;
    prefactor_BL      *=2;
    prefactor_BA      *=2;
    prefactor_LJ_wall *=2;
    prefactor_el_x    *=2;
    prefactor_el_y    *=2;
    prefactor_el_z    *=2;
  }
// Create output-strings for easier analysis
  create_output_strings();
}

void displayconstants (char* stdout_filename) {
  FILE* stdout_file = fopen(stdout_filename, "a");
  fprintf(stdout_file, "# use_verlet       = %i\n", use_verlet);
  fprintf(stdout_file, "# enable_LJ        = %i\n", enable_LJ);
  fprintf(stdout_file, "# backbone_LJ      = %i\n", backbone_LJ);
  fprintf(stdout_file, "# enable_BL        = %i\n", enable_BL);
  fprintf(stdout_file, "# enable_BA        = %i\n", enable_BA);
  fprintf(stdout_file, "# enable_BM        = %i\n", enable_BM);
  fprintf(stdout_file, "# enable_FR        = %i\n", enable_FR);
  fprintf(stdout_file, "# enable_SP        = %i\n", enable_SP);
  fprintf(stdout_file, "# enable_trap      = %i\n", enable_trap);
  fprintf(stdout_file, "# enable_mirror_trap   = %i\n", enable_mirror_trap);
  fprintf(stdout_file, "# enable_wall_y      = %i\n", enable_wall_y);
  fprintf(stdout_file, "# enable_el        = %i\n", enable_el);
  fprintf(stdout_file, "# enable_el_trap     = %i\n", enable_el_trap);
  fprintf(stdout_file, "# enable_pulse_el    = %i\n", enable_pulse_el);
  fprintf(stdout_file, "# display_mode_1     = %i\n", display_mode_1);
  fprintf(stdout_file, "# display_gyration    = %i\n", display_gyration);
  fprintf(stdout_file, "# display_end_to_end   = %i\n", display_end_to_end);
  fprintf(stdout_file, "# display_min_max     = %i\n", display_min_max);
  fprintf(stdout_file, "# display_min_max_mon   = %i\n", display_min_max_mon);
  fprintf(stdout_file, "# display_min_max_all   = %i\n", display_min_max_all);
  fprintf(stdout_file, "# display_contour     = %i\n", display_contour);
  fprintf(stdout_file, "# display_min_max_bond  = %i\n", display_min_max_bond);
  fprintf(stdout_file, "# display_mean_speed   = %i\n", display_mean_speed);
  fprintf(stdout_file, "# display_ang_momentum  = %i\n", display_ang_momentum);
  fprintf(stdout_file, "# display_energy      = %i\n", display_energy);
  fprintf(stdout_file, "# display_split_energy  = %i\n", display_split_energy);
  fprintf(stdout_file, "# display_cos_theta    = %i\n", display_cos_theta);
  fprintf(stdout_file, "# display_bond_correl   = %i\n", display_bond_correl);
  fprintf(stdout_file, "# display_current_factor = %i\n", display_bond_correl);
  fprintf(stdout_file, "# append_configs      = %i\n", append_configs);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# steps          = %i\n", steps);
  fprintf(stdout_file, "# output_steps      = %i\n", output_steps);
  fprintf(stdout_file, "# config_steps      = %i\n", config_steps);
  fprintf(stdout_file, "# throw_away       = %i\n", throw_away);
  fprintf(stdout_file, "# delta_t         = %lf\n", delta_t);
```

constants.cc

```
  fprintf(stdout_file, "# sqrt_delta_t      = %lf\n", sqrt_delta_t);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# zeta_FR          = %lf\n", zeta_FR);
  fprintf(stdout_file, "# decay_v          = %lf\n", decay_v);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# T              = %lf\n", T);
  fprintf(stdout_file, "# sqrt_T           = %lf\n", sqrt_T);
  fprintf(stdout_file, "# epsilon_BM       = %lf\n", epsilon_BM);
  fprintf(stdout_file, "# prefactor_BM     = %lf\n", prefactor_BM);
  fprintf(stdout_file, "# rand-seed        = %u\n", rand_seed);
  fprintf(stdout_file, "# save-new-seed      = %u\n", save_new_seed);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_SP       = %lf\n", epsilon_SP);
  fprintf(stdout_file, "# prefactor_SP     = %lf\n", prefactor_SP);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_LJ       = %lf\n", epsilon_LJ);
  fprintf(stdout_file, "# sigma_LJ         = %lf\n", sigma_LJ);
  fprintf(stdout_file, "# v_cutoff_LJ      = %lf\n", v_cutoff_LJ);
  fprintf(stdout_file, "# l_cutoff_LJ      = %lf\n", l_cutoff_LJ);
  fprintf(stdout_file, "# l_cutoff_LJ_sqr    = %lf\n", l_cutoff_LJ_sqr);
  fprintf(stdout_file, "# sigma_6          = %lf\n", sigma_6);
  fprintf(stdout_file, "# prefactor_LJ     = %lf\n", prefactor_LJ);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_BL       = %lf\n", epsilon_BL);
  fprintf(stdout_file, "# d_BL           = %lf\n", d_BL);
  fprintf(stdout_file, "# d_0_BL         = %lf\n", d_0_BL);
  fprintf(stdout_file, "# d_BL_sqr         = %lf\n", d_BL_sqr);
  fprintf(stdout_file, "# prefactor_BL     = %lf\n", prefactor_BL);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_BA       = %lf\n", epsilon_BA);
  fprintf(stdout_file, "# prefactor_BA     = %lf\n", prefactor_BA);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_LJ_wall    = %lf\n", epsilon_LJ_wall);
  fprintf(stdout_file, "# sigma_LJ_wall      = %lf\n", sigma_LJ_wall);
  fprintf(stdout_file, "# sigma_6_wall      = %lf\n", sigma_6_wall);
  fprintf(stdout_file, "# l_cutoff_LJ_wall    = %lf\n", l_cutoff_LJ_wall);
  fprintf(stdout_file, "# l_cutoff_LJ_wall_sqr  = %lf\n", l_cutoff_LJ_wall_sqr);
  fprintf(stdout_file, "# v_cutoff_LJ_wall    = %lf\n", v_cutoff_LJ_wall);
  fprintf(stdout_file, "# prefactor_LJ_wall   = %lf\n", prefactor_LJ_wall);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# h_top_trap       = %lf\n", h_top_trap);
  fprintf(stdout_file, "# h_bottom_trap      = %lf\n", h_bottom_trap);
  fprintf(stdout_file, "# l_bottom_trap      = %lf\n", l_bottom_trap);
  fprintf(stdout_file, "# l_top_trap       = %lf\n", l_top_trap);
  fprintf(stdout_file, "# wall_tilt_trap     = %lf\n", wall_tilt_trap);
  fprintf(stdout_file, "# l_surface_trap     = %lf\n", l_surface_trap);
  fprintf(stdout_file, "# h_total_trap       = %lf\n", h_total_trap);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# wall_y           = %lf\n", wall_y);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_el_x       = %lf\n", epsilon_el_x);
  fprintf(stdout_file, "# epsilon_el_y       = %lf\n", epsilon_el_y);
  fprintf(stdout_file, "# epsilon_el_z       = %lf\n", epsilon_el_z);
  fprintf(stdout_file, "# prefactor_el_x     = %lf\n", prefactor_el_x);
  fprintf(stdout_file, "# prefactor_el_y     = %lf\n", prefactor_el_y);
  fprintf(stdout_file, "# prefactor_el_z     = %lf\n", prefactor_el_z);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# epsilon_el_trap    = %lf\n", epsilon_el_trap);
  fprintf(stdout_file, "# potential_file_trap  %s\n", potential_file_trap);
  fprintf(stdout_file, "# trap_scaling_factor  = %lf\n", trap_scaling_factor);
  fprintf(stdout_file, "# trap_discretize     = %lf\n", trap_discretize);
  fprintf(stdout_file, "# trap_max_x       = %i\n", trap_max_x);
  fprintf(stdout_file, "# trap_max_z       = %i\n", trap_max_z);
  fprintf(stdout_file, "# trap_top_x       = %i\n", trap_top_x);
  fprintf(stdout_file, "# trap_top_z       = %i\n", trap_top_z);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# pulse_period_el    = %i\n", pulse_period_el);
  fprintf(stdout_file, "# pulse_uptime_el    = %i\n", pulse_uptime_el);
  fprintf(stdout_file, "# pulse_factor_el    = %lf\n", pulse_factor_el);
  fprintf(stdout_file, "# \n");
  fprintf(stdout_file, "# displayed_variables = %i\n", displayed_variables);
  fprintf(stdout_file, "# ");
  for (int i = 1; i <= displayed_variables; i++)
    fprintf(stdout_file, "%s[%i] ", stdout_component[i], i);
  fprintf(stdout_file, "\n");
  fprintf(stdout_file, "# \n");
  fclose(stdout_file);
}

void write_constants(char* filename) {
  FILE* input = fopen(filename, "r");
  if (input) {                                          // Do not overwrite an existing file
    cerr << "Modified-constants-file already exists!\n";
    exit(17);
  }
  FILE* output = fopen(filename, "w");
  fprintf(output, "version        7\n");
  fprintf(output, "use_verlet       %i\n", use_verlet);
  fprintf(output, "enable_LJ        %i\n", enable_LJ);
  fprintf(output, "backbone_LJ      %i\n", backbone_LJ);
  fprintf(output, "enable_BL        %i\n", enable_BL);
  fprintf(output, "enable_BA        %i\n", enable_BA);
  fprintf(output, "enable_FR        %i\n", enable_FR);
  fprintf(output, "enable_BM        %i\n", enable_BM);
  fprintf(output, "enable_SP        %i\n", enable_SP);
  fprintf(output, "enable_trap      %i\n", enable_trap);
  fprintf(output, "enable_mirror_trap   %i\n", enable_mirror_trap);
  fprintf(output, "enable_wall_y    %i\n", enable_wall_y);
  fprintf(output, "enable_el      %i\n", enable_el);
  fprintf(output, "enable_el_trap     %i\n", enable_el_trap);
  fprintf(output, "enable_pulse_el    %i\n", enable_pulse_el);
  fprintf(output, "display_mode_1     %i\n", display_mode_1);
  fprintf(output, "display_gyration    %i\n", display_gyration);
```

constants.cc

```
  fprintf(output, "display_end_to_end    %i\n", display_end_to_end);
  fprintf(output, "display_min_max      %i\n", display_min_max);
  fprintf(output, "display_min_max_mon   %i\n", display_min_max_mon);
  fprintf(output, "display_min_max_all   %i\n", display_min_max_all);
  fprintf(output, "display_contour      %i\n", display_contour);
  fprintf(output, "display_min_max_bond  %i\n", display_min_max_bond);
  fprintf(output, "display_mean_speed    %i\n", display_mean_speed);
  fprintf(output, "display_ang_momentum  %i\n", display_ang_momentum);
  fprintf(output, "display_energy       %i\n", display_energy);
  fprintf(output, "display_split_energy  %i\n", display_split_energy);
  fprintf(output, "display_cos_theta     %i\n", display_cos_theta);
  fprintf(output, "display_bond_correl   %i\n", display_bond_correl);
  fprintf(output, "display_current_factor %i\n", display_bond_correl);
  fprintf(output, "display_constants     %i\n", display_constants);
  fprintf(output, "\n");
  fprintf(output, "Number_of_steps      %i\n", steps);
  fprintf(output, "Output_steps        %i\n", output_steps);
  fprintf(output, "Config_steps        %i\n", config_steps);
  fprintf(output, "throw_away          %i\n", throw_away);
  fprintf(output, "Time-step          %lf\n", delta_t);
  fprintf(output, "\n");
  fprintf(output, "zeta_FR           %lf\n", zeta_FR);
  fprintf(output, "\n");
  fprintf(output, "Temperature         %lf\n", T);
  fprintf(output, "epsilon_BM          %lf\n", epsilon_BM);
  fprintf(output, "save_new_seed       %i\n", save_new_seed);
  fprintf(output, "\n");
  fprintf(output, "epsilon_SP        %lf\n", epsilon_SP);
  fprintf(output, "\n");
  fprintf(output, "epsilon_LJ        %lf\n", epsilon_LJ);
  fprintf(output, "sigma_LJ          %lf\n", sigma_LJ);
  fprintf(output, "\n");
  fprintf(output, "epsilon_BL        %lf\n", epsilon_BL);
  fprintf(output, "d_BL             %lf\n", d_BL);
  fprintf(output, "d_0_BL           %lf\n", d_0_BL);
  fprintf(output, "\n");
  fprintf(output, "epsilon_BA        %lf\n", epsilon_BA);
  fprintf(output, "\n");
  fprintf(output, "epsilon_LJ_wall     %lf\n", epsilon_LJ_wall);
  fprintf(output, "sigma_LJ_wall       %lf\n", sigma_LJ_wall);
  fprintf(output, "l_cutoff_LJ_wall    %lf\n", l_cutoff_LJ_wall/sigma_LJ_wall);
  fprintf(output, "\n");
  fprintf(output, "h_top_trap        %lf\n", h_top_trap);
  fprintf(output, "h_bottom_trap      %lf\n", h_bottom_trap);
  fprintf(output, "l_bottom_trap      %lf\n", l_bottom_trap);
  fprintf(output, "l_top_trap        %lf\n", l_top_trap);
  fprintf(output, "wall_tilt_trap      %lf\n", wall_tilt_trap);
  fprintf(output, "\n");
  fprintf(output, "wall_y           %lf\n", wall_y);
  fprintf(output, "\n");
  fprintf(output, "epsilon_el_x       %lf\n", epsilon_el_x);
  fprintf(output, "epsilon_el_y       %lf\n", epsilon_el_y);
  fprintf(output, "epsilon_el_z       %lf\n", epsilon_el_z);
  fprintf(output, "\n");
  fprintf(output, "epsilon_el_trap     %lf\n", epsilon_el_trap);
  fprintf(output, "potential_file_trap  %s\n", potential_file_trap);
  fprintf(output, "\n");
  fprintf(output, "pulse_period_el     %i\n", pulse_period_el);
  fprintf(output, "pulse_uptime_el     %i\n", pulse_uptime_el);
  fprintf(output, "pulse_factor_el     %lf\n", pulse_factor_el);
  fprintf(output, "\n");
  fclose(output);
}

#endif
```

constants.hh

```
#include<iostream>
#include<fstream>
#include<string>
#include<math.h>
#include<stdio.h>

#ifndef CONSTANTS_INCLUDED
#define CONSTANTS_INCLUDED

using namespace std;

// This file reads the constants for the main program and a function for displaying
// them, if desired.

// These are the switches for the potentials, algorithms and output
extern int use_verlet;
extern int enable_LJ;
extern int backbone_LJ;
extern int enable_BL;
extern int enable_BA;
extern int enable_FR;
extern int enable_BM;
extern int enable_SP;
extern int enable_trap;
extern int enable_mirror_trap;
extern int enable_wall_y;
extern int enable_el;
extern int enable_el_trap;
extern int enable_pulse_el;
extern int display_constants;
extern int display_min_max;
extern int display_min_max_mon;
extern int display_min_max_all;
extern int display_contour;
extern int display_min_max_bond;
extern int display_gyration;
extern int display_end_to_end;
extern int display_mode_1;
extern int display_mean_speed;
extern int display_ang_momentum;
extern int display_energy;
extern int display_split_energy;
extern int display_cos_theta;
extern int display_bond_correl;
extern int display_current_factor;
extern int displayed_variables;
extern int save_new_seed;
extern int append_configs;

// This is the number of steps
extern int steps;
// This is the number of displayed configurations
extern int output_steps;
// This is the number of displayed configurations in full
extern int config_steps;
// This is the number of thwrow-away-steps for initialisation
extern int throw_away;

// This is the time-step
extern double delta_t;
extern double sqrt_delta_t;

// This is the friction-constant
extern double zeta_FR;
extern double decay_v;

// This is the temperature
extern double T;
extern double sqrt_T;

// This is for the brownian motion
extern double epsilon_BM;
extern double prefactor_BM;
extern unsigned long rand_seed;

// This is the spring-constant
extern double epsilon_SP;
extern double prefactor_SP;

// These are the constants for the Lennard-Jones potential between two monomers
extern double epsilon_LJ;
extern double sigma_LJ;
extern double v_cutoff_LJ;
extern double l_cutoff_LJ;
extern double l_cutoff_LJ_sqr;
extern double sigma_6;
extern double prefactor_LJ;

// These are the constants for the bond-length-energy between two monomers
// on the chain
extern double epsilon_BL;
extern double d_BL;
extern double d_BL_sqr;
extern double d_0_BL;
extern double prefactor_BL;

// This is the constant for the bond-angle-energy
extern double epsilon_BA;
extern double prefactor_BA;

//This is the wall Lennard-Jones-potential
extern double epsilon_LJ_wall;
extern double sigma_LJ_wall;
```

constants.hh

```
extern double sigma_6_wall;
extern double l_cutoff_LJ_wall;
extern double l_cutoff_LJ_wall_sqr;
extern double v_cutoff_LJ_wall;
extern double prefactor_LJ_wall;

//This is the entropic trap/structured microchannel
extern double h_top_trap;
extern double h_bottom_trap;
extern double l_bottom_trap;
extern double l_top_trap;
extern double wall_tilt_trap;
extern double l_surface_trap;
extern double h_total_trap;

//This is the wall in y-direction for entropic trap
extern double wall_y;

// This is the homogeneous external electric field
extern double epsilon_el_x;
extern double epsilon_el_y;
extern double epsilon_el_z;
extern double prefactor_el_x;
extern double prefactor_el_y;
extern double prefactor_el_z;

//This is the electrical field inside the trap
extern double epsilon_el_trap;
extern char* potential_file_trap;
extern int trap_max_x;
extern int trap_max_z;
extern int trap_top_x;
extern int trap_top_z;
extern double trap_scaling_factor;
extern double trap_discretize;

//This the pulsed electrical field
extern int pulse_period_el;      // Pulse period(=(uptime+downtime) in timesteps
extern int pulse_uptime_el;      // Uptime begins at beginning of period
extern double pulse_factor_el;   // Rescale-factor in the downtime
extern double current_factor_el; // Changes during simulation between 1 and Rescale-factor

// This is for analyzing the output-files
extern char** stdout_component;

// ------------------------------------------------------------------------

int create_output_strings(void);

void init_constants(char* filename);

void displayconstants (char* stdout_filename);

void write_constants(char* filename);

#endif
```

errorlist.txt

```
-------------------------------------------------------------------------------------------------------------
Code    Module          Error
-------------------------------------------------------------------------------------------------------------
--- Initialisation errors ---------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
1       main_polymer.cc         Invalid number of Arguments
2       constants.cc            Constants-file not found
3       constants.cc            Wrong version of constants-file
4       constants.cc            Random-seed not found
5       verlet.cc               Invalid start-file name
6       verlet.cc               Too few monomers in start-file
7       verlet.cc               Start- and checkpoint are identical and not all monomers used
8       verlet.cc               Checkpoint-file exists and startfile is different
9       constants.cc            Potential-file not fond
10      constants.cc            Potential-file and/or geometry do not fit
11      constants.cc            Incompatible step-sizes
18      constants.cc            Incompatible switches (Field_trap enabled, trap disabled)
19      config_utils.cc         Incomplete checkpoint
20      constants.cc            Tilt trap is only allowed non-mirrored
21      config_utils.cc         Throw away may only be applied with old timesteps == 0!
-------------------------------------------------------------------------------------------------------------
--- Errors that should not happen -------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
12      ipoint_map.cc           Error: Box-references are faulty (reference == 0)
13      ipoint_map.cc           Invalid access in fast_LJ_update - cube is not in map
-------------------------------------------------------------------------------------------------------------
--- Run-time errors that may occur during simulation ------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
14      verlet.cc               Rupture in    verlet-timestep - bond longer than cutoff-potential
15      verlet.cc               Rupture in no-verlet-timestep - bond longer than cutoff-potential
-------------------------------------------------------------------------------------------------------------
--- Analysis errors ---------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
16      main_analyse_rouse.cc   Diffusion can only be determined using a finite time-step
17      constants.cc            Modified-constants-file already exists
-------------------------------------------------------------------------------------------------------------
```

ipoint.hh

```
#include<math.h>
#include<iostream>
#include"point.hh"

#ifndef IPOINT_INCLUDED
#define IPOINT_INCLUDED

using namespace std;

// This is a class of 3-dimensional integers. It includes a conversion-operator
// ipoint(double, double, double) and also ipoint(point) for easy conversion of
// 3-dimensional points. It is necessary to include "point.hpp" first in the
// main program. It also includes order-operators (<, >, <=, >=) which first
// sort the ipoints by the first component, then by the second component and
// at last by the third entry. This is used for sorting the ipoints in the
// index-map.

class ipoint {
  int x; int y; int z;
public:
  ipoint()
  {}
  ipoint(int xx, int yy, int zz){
    x = xx; y = yy; z = zz;
  }
  ipoint(double xx, double yy, double zz){
    x = int(std::floor(xx)); y = int(std::floor(yy)); z = int(std::floor(zz));
  }
  ipoint(point p){
    x = int(std::floor(p.component(1)));
    y = int(std::floor(p.component(2)));
    z = int(std::floor(p.component(3)));
  }
  friend ostream& operator<<(ostream& os, ipoint p){
    return os << "(" << p.x << "," << p.y << "," << p.z << ")";
  }
  friend bool operator<(ipoint p1, ipoint p2){
    if ((p1.x == p2.x) && (p1.y == p2.y)) return (p1.z < p2.z);
    if (p1.x == p2.x) return (p1.y < p2.y);
    return (p1.x < p2.x);
  }
  friend bool operator>(ipoint p1, ipoint p2){
    return (p2 < p1);
  }
  friend bool operator<=(ipoint p1, ipoint p2){
    return (p1 == p2) || (p1 < p2);
  }
  friend bool operator>=(ipoint p1, ipoint p2){
    return (p1 == p2) || (p1 > p2);
  }
  friend bool operator==(ipoint p1, ipoint p2){
    return ((p1.x==p2.x) && (p1.y==p2.y) && (p1.z==p2.z));
  }
  friend bool operator!=(ipoint p1, ipoint p2){
    return !(p1==p2);
  }
  friend ipoint operator+(ipoint p1, ipoint p2){
    return ipoint(p1.x + p2.x, p1.y + p2.y, p1.z + p2.z);
  }
  ipoint operator+=(ipoint p){
    x += p.x; y += p.y; z += p.z;
    return ipoint(x,y,z);
  }
  friend ipoint operator-(ipoint p){
    return ipoint(-p.x, -p.y, -p.z);
  }
  friend ipoint operator-(ipoint p1, ipoint p2){
    return ipoint(p1.x-p2.x, p1.y-p2.y, p1.z-p2.z);
  }
  ipoint operator-=(ipoint p){
    x -= p.x; y -= p.y; z -= p.z;
    return ipoint(x,y,z);
  }
  int component(int choice){
    return (choice==1 ? x : (choice==2 ? y : z));
  }
  int x_(void){
    return x;
  }
  int y_(void){
    return y;
  }
  int z_(void){
    return z;
  }
};

#endif
```

ipoint_map.cc

```
#include<iostream>
#include<map>
#include<stdio.h>
#include"ipoint.hh"
#include"point.hh"
#include"potentials.hh"
#include"constants.hh"

#ifndef MAP_IPOINT_INCLUDED
#define MAP_IPOINT_INCLUDED

using namespace std;

// This is the implementation of the red-black-tree containing the indices
// of each cube. It contains functions for inserting/ removing indices
// (the appropriate cubes are inserted/ deleted automatically to save
// memory), displaying a certain cube plus its indices and neighbours,
// counting the indices in a certain cube and functions to create an
// array containing all indices within a cube and of a certain amount
// of neighbours. For the nearest neighbours and an included center-
// cube, a specialized function for readout exists. It gains speed by
// accessing the neighbours directly through the references.
//
// In many functions, a pointer 'mouse' is used. It is used to navigate within
// the tree, like a computer-mouse is used to navigate on the screen.
//
// Structures used:
//   - node : contains the current cube, the number of indices in the current
//            cube, an array of indices, the size of the array, the reference
//            -field to the nearest neighbours and a standard-constructor
//
// Constants used:
//   - max_neighbours  : the maximum number of nearest neighbours of a cube
//                       (equals 3*3*3, a cube with side-length 3)
//
//  Functions used (functions in brackets are not intended to be called directly
//                  by the user and are not in the header-file):
//   - (create_new) : This function creates a new node. Stores the cube in the
//                    node, creates an array for two indices, sets the number
//                    of valid entries to zero. Creates the reference-array
//                    and fills it with the valid references. Updates the
//                    refrences of the neighbouring cubes as well.
//   - map_insert_index : Inserts an index in the red-black-tree.
//                        If necessary, it creates a new node for the cube.
//                        Checks, if the index-array is big enough to store
//                        another index. If not, the size of the index is
//                        doubled. Stores the new index at the end of the array
//                        and increases the number of valid entries by one.
//   - map_output : Displays the desired cube, and, if valid, the indices within
//                  the cube and the valid nearest-neighbour cubes.
//   - (destroy) : This function destroys a node. Deletes the references and the
//                 backpointing ones (from the neighbours) as well. Deletes the
//                 index-array and at last the node itself.
//   - map_remove_index : Removes an index from the red-black-tree. Checks, if the cube
//                        is valid and contains the correct index. Swaps the index
//                        to be deleted to the end of the field. Decreases the number
//                        of valid entries by one. If no valid entries are left,
//                        the node is destroyed.
//   - index_count : Counts the indices within a certain cube. Returns zero for
//                   a non-existing cube.
//   - indices_in_cube : Returns an array containing all valid indices within cube.
//                       Creates a new integer-array, so don't forget to free the
//                       memory for it afterwards. Also returns the number of
//                       valid indices.
//   - old_neighbours_of_cube : Returns an array containing the indices in the
//                              cube and those of the nearest neighbours (default
//                              is one). This function is non-recursive, but slow due
//                              to many scans of the red-black-tree.
//   - neighbours_of_cube : Checks, if cube is located in the red-black-tree and if the number
//                          of desired nearest neighbours is set to one (the default).
//                          If not, a warning is written on stderr and old_neighbours_of_cube
//                          is called, it is able to handle these difficulties. If the
//                          check is OK, it accesses the current cube directly and
//                          the nearest neighbours by the reference-list, which is in
//                          each node and thus saving a lot of time.
//   - fast_LJ_update : Scans through the tree for all cubes, and evaluates the Lennard-Jones-
//                      interaction potential. Almost similar to neighbours_of_cube and
//                      LJ_update, but avoides copying of indices.

struct node {
  ipoint cube;                    // stores the (lattice-)cube
  int max_indices_in_cube;        // the size of the index-array
  int indices_in_cube;            // the number of indices in the cube
  int* index;                     // location of the indices within the cube (array)
  node ** reference;              // stores the references to the nearest neighbours
  int number_of_references;       // the number of occupied references
  node() {}                       // Default-constructor
};

// Maximum number of nearest neighbours of a cube
const int max_neighbours = 27;

typedef map<ipoint, node*> map_ipoint;

node* create_new(map_ipoint &idx_map, ipoint cube) {
  node* mouse = new node;                     // create the new node
  mouse->cube = cube;
  mouse->max_indices_in_cube = 2;             // create an array for 2 indices
  mouse->indices_in_cube = 0;                 // and set the number of valid entries to 0
  mouse->index = new int[mouse->max_indices_in_cube];
  mouse->reference = new (node*)[max_neighbours];
  mouse->number_of_references = 0;
  for (int i = 0; i < max_neighbours; i++)
```

ipoint_map.cc

```
    mouse->reference[i] = 0;                        // scan the tree for all nearest neighbours
  for (int dx = -1; dx <= 1; dx++)
    for (int dy = -1; dy <= 1; dy++)
      for (int dz = -1; dz <= 1; dz++)
        if (ipoint(dx, dy, dz) != ipoint(0,0,0)) {
          map_ipoint::iterator search = idx_map.find(ipoint(dx, dy, dz) + cube);
          if (search != idx_map.end()) {
            node* search_mouse = idx_map[ipoint(dx, dy, dz) + cube];
            mouse->reference[mouse->number_of_references] = search_mouse;
            mouse->number_of_references++;
            search_mouse->reference[search_mouse->number_of_references] = mouse;
            search_mouse->number_of_references++;
          }
        }
  return mouse;
}

void map_insert_index(map_ipoint& idx_map, ipoint cube, node*& box_ref, int index) {
  node* mouse = idx_map[cube];
  if (mouse == 0) {                                 // cube is not in the map
    mouse = create_new(idx_map, cube);              // and has to be created
    idx_map[cube] = mouse;
  }
  if (mouse->indices_in_cube == mouse->max_indices_in_cube) {
    mouse->max_indices_in_cube *= 2;
    int* new_index = new int[mouse->max_indices_in_cube];  // create a new array capable
    for(int i = 0; i < mouse->indices_in_cube; i++)        // of storing all old indices
      new_index[i] = mouse->index[i];                      // and the new one
    {
      int * tmp = mouse->index;                          // swap index-fields
      mouse->index = new_index;
      new_index = tmp;
    }
    delete [] new_index;
  }
  mouse->index[mouse->indices_in_cube] = index;         // enter the new index into the array
  mouse->indices_in_cube++;
  box_ref = mouse;
}

void map_output(map_ipoint& idx_map, ipoint cube, char* stdout_filename) {
  FILE* stdout_file = fopen(stdout_filename, "a");
  node* mouse = idx_map[cube];
  if (mouse == 0) {
    fprintf(stdout_file, ">> (%i,%i,%i) <<\n", cube.x_(), cube.y_(), cube.z_());
    idx_map.erase(cube);
  } else {
    fprintf(stdout_file, "(%i,%i,%i) {", cube.x_(), cube.y_(), cube.z_());
    for (int i = 0; i < mouse->indices_in_cube; i++)
      fprintf(stdout_file, "%i,", mouse->index[i]);
    fprintf(stdout_file, "%c}, ", 8);
    fprintf(stdout_file, "%i Neighbours: >> ", mouse->number_of_references);
    for (int i = 0; i < mouse->number_of_references; i++)
      if (mouse->reference[i] != 0)
        fprintf(stdout_file, "(%i,%i,%i)", mouse->reference[i]->cube.x_(), mouse->reference[i]->cube.y_(),
                mouse->reference[i]->cube.z_());
    fprintf(stdout_file, "<<\n");
  }
  fclose(stdout_file);
}

void destroy(node*& mouse) {
// This function actually deletes the mouse, also deletes the references
// in both directions
  for (int i = 0; i < mouse->number_of_references; i++) {               // scan all valid references
    for (int j = 0; j < mouse->reference[i]->number_of_references; j++) // locate the back-pointing reference
      if (mouse->reference[i]->reference[j] == mouse) {
        mouse->reference[i]->reference[j] =
          mouse->reference[i]->reference[(mouse->reference[i]->number_of_references)-1]; // and delete it
        (mouse->reference[i]->number_of_references)--;
      }
  }
  delete [] mouse->reference;                                       // delete all pointers and arrays
  delete [] mouse->index;
  delete mouse;
}

void map_remove_index(map_ipoint& idx_map, ipoint cube, node*& box_ref, int index) {
  node* mouse = idx_map[cube];
  if (mouse == 0) {                                 // cube has to be within the map
    idx_map.erase(cube);
    cerr << "Invalid Deletion - cube is not in tree!\n";
  } else {
    bool found = false;
    for (int i = 0; i<mouse->indices_in_cube; i++)    // search the right index
      if (mouse->index[i] == index) {
        found = true;
        int tmp = mouse->index[i];                                  // swap the index which is to be
        mouse->index[i] = mouse->index[mouse->indices_in_cube-1]; // deleted to the end of the field
        mouse->index[mouse->indices_in_cube-1] = tmp;
        i = mouse->indices_in_cube;
      }
    if (!found) {
      cerr << "Invalid Deletion - index is not in cube!\n";
    } else {
      mouse->indices_in_cube--;           // and reduce the number of valid indices by one
    }
    if (mouse->indices_in_cube == 0) {    // remove the cube, if necessary
      destroy(mouse);
      idx_map.erase(cube);
    }
  }
  box_ref = 0;
```

ipoint_map.cc

```
}

int index_count(node*& box_ref)
// Counts the number of indices within the node, returns 0 when cube is not in
// the red-black-tree.
{
  if (box_ref == 0) {
    return 0;
  } else {
    return box_ref->indices_in_cube;
  }
}

int* indices_in_cube(node*& box_ref, int& number)
// Returns an array of the indices located in cube. Does not return empty
// indices. The size of the array is also returned in number. This function
// allocates the memory for the array, so don't forget to free the memory
// after usage.
{
  if (box_ref == 0) {
    number = 0;
    return 0;
  } else {
    number = box_ref->indices_in_cube;
    int* idx_list = new int[number];
    for (int i = 0; i < number; i++)
      idx_list[i] = box_ref->index[i];
    return idx_list;
  }
}

int* neighbours_of_cube(node*& box_ref, int& number) {
// This function checks if the cube is located in the tree and if only
// the first nearest neighbours (default) are desired. If one of these
// checks fail, a warning is printed on stderr and old_neighbours_of_cube
// is called, which is capable of handling these difficulties, but slow.
// If these tests are successfully passed, it reads out the indices in the
// current cube and its nearest neighbours, which are accessed directly
// via the reference-list.
  if (box_ref == 0) {
    cerr << "Error: Box-references are faulty!\n";
    exit(12);
  }
  number = box_ref->indices_in_cube;                    // count the number of indices to be returned
  for (int i = 0; i < box_ref->number_of_references; i++)
    number += box_ref->reference[i]->indices_in_cube;
  int* idx_list = new int[number];
  number = 0;
  for (int i = 0; i < box_ref->indices_in_cube; i++) {
    idx_list[number] = box_ref->index[i];
    number++;
  }
  for (int i = 0; i < box_ref->number_of_references; i++)
    for (int j = 0; j < box_ref->reference[i]->indices_in_cube; j++) {
      idx_list[number] = box_ref->reference[i]->index[j];
      number++;
    }
  return idx_list;
}

void fast_LJ_update(point* r, point* dv, node**& box_ref, int N) {
// Scans through the tree for all indices, evaluates the Lennard-Jones-interaction.
// Almost similar to neighbours_of_cube and LJ_update, but avoides copying of indices.
  for (int i = 0; i < N; i++) {
    node* mouse = box_ref[i];
    if (mouse == 0) {                                   // if it is not found, exit the program
      cerr << "Invalid access in fast_LJ_update - cube is not in red-black-tree!\n";
      exit(13);
    }
    for (int j = 0; j < mouse->indices_in_cube; j++) { // scan the cube in the center
      int index = mouse->index[j];
      if ((index < i) && (backbone_LJ || (i-index != 1))) {
        point d_V_LJ = d_V_LJ_dr_times_timestep(r[index] - r[i]);
        dv[i]     +=  d_V_LJ;
        dv[index] += -d_V_LJ;
      }
    }
    for (int k = 0; k < mouse->number_of_references; k++) {  // and all neighbours through the references
      node* search_mouse = mouse->reference[k];
      for (int j = 0; j < search_mouse->indices_in_cube; j++) {
        int index = search_mouse->index[j];
        if ((index < i) && (backbone_LJ || (i-index != 1))) {
          point d_V_LJ = d_V_LJ_dr_times_timestep(r[index] - r[i]);
          dv[i]     +=  d_V_LJ;
          dv[index] += -d_V_LJ;
        }
      }
    }
  }
}

#endif
```

ipoint_map.hh

```
#include<iostream>
#include<map>
#include<stdio.h>
#include"ipoint.hh"
#include"point.hh"
#include"potentials.hh"
#include"constants.hh"

#ifndef MAP_IPOINT_INCLUDED
#define MAP_IPOINT_INCLUDED

using namespace std;

// This is the implementation of the red-black-tree containing the indices
// of each cube. It contains functions for inserting/ removing indices
// (the appropriate cubes are inserted/ deleted automatically to save
// memory), displaying a certain cube plus its indices and neighbours,
// counting the indices in a certain cube and functions to create an
// array containing all indices within a cube and of a certain amount
// of neighbours. For the nearest neighbours and an included center-
// cube, a specialized function for readout exists. It gains speed by
// accessing the neighbours directly through the references.
//
// In many functions, a pointer 'mouse' is used. It is used to navigate within
// the tree, like a computer-mouse is used to navigate on the screen.
//
// Structures used:
//   - node : contains the current cube, the number of indices in the current
//            cube, an array of indices, the size of the array, the reference
//            -field to the nearest neighbours and a standard-constructor
//
// Constants used:
//   - max_neighbours  : the maximum number of nearest neighbours of a cube
//                       (equals 3*3*3, a cube with side-length 3)
//
//  Functions used (functions in brackets are not intended to be called directly
//                  by the user and are not in the header-file):
//   - (create_new) : This function creates a new node. Stores the cube in the
//                    node, creates an array for two indices, sets the number
//                    of valid entries to zero. Creates the reference-array
//                    and fills it with the valid references. Updates the
//                    refrences of the neighbouring cubes as well.
//   - map_insert_index : Inserts an index in the red-black-tree.
//                        If necessary, it creates a new node for the cube.
//                        Checks, if the index-array is big enough to store
//                        another index. If not, the size of the index is
//                        doubled. Stores the new index at the end of the array
//                        and increases the number of valid entries by one.
//   - map_output : Displays the desired cube, and, if valid, the indices within
//                  the cube and the valid nearest-neighbour cubes.
//   - (destroy) : This function destroys a node. Deletes the references and the
//                 backpointing ones (from the neighbours) as well. Deletes the
//                 index-array and at last the node itself.
//   - map_remove_index : Removes an index from the red-black-tree. Checks, if the cube
//                        is valid and contains the correct index. Swaps the index
//                        to be deleted to the end of the field. Decreases the number
//                        of valid entries by one. If no valid entries are left,
//                        the node is destroyed.
//   - index_count : Counts the indices within a certain cube. Returns zero for
//                   a non-existing cube.
//   - indices_in_cube : Returns an array containing all valid indices within cube.
//                       Creates a new integer-array, so don't forget to free the
//                       memory for it afterwards. Also returns the number of
//                       valid indices.
//   - old_neighbours_of_cube : Returns an array containing the indices in the
//                              cube and those of the nearest neighbours (default
//                              is one). This function is non-recursive, but slow due
//                              to many scans of the red-black-tree.
//   - neighbours_of_cube : Checks, if cube is located in the red-black-tree and if the number
//                          of desired nearest neighbours is set to one (the default).
//                          If not, a warning is written on stderr and old_neighbours_of_cube
//                          is called, it is able to handle these difficulties. If the
//                          check is OK, it accesses the current cube directly and
//                          the nearest neighbours by the reference-list, which is in
//                          each node and thus saving a lot of time.
//   - fast_LJ_update : Scans through the tree for all cubes, and evaluates the Lennard-Jones-
//                      interaction potential. Almost similar to neighbours_of_cube and
//                     LJ_update, but avoides copying of indices.

struct node {
  ipoint cube;                    // stores the (lattice-)cube
  int max_indices_in_cube;        // the size of the index-array
  int indices_in_cube;            // the number of indices in the cube
  int* index;                     // location of the indices within the cube (array)
  node ** reference;              // stores the references to the nearest neighbours
  int number_of_references;       // the number of occupied references
  node() {}                       // Default-constructor
};

typedef map<ipoint, node*> map_ipoint;

void map_insert_index(map_ipoint& idx_map, ipoint cube, node*& box_ref, int index);

void map_output(map_ipoint& idx_map, ipoint cube, char* stdout_filename);

void map_remove_index(map_ipoint& idx_map, ipoint cube, node*& box_ref, int index);

int* neighbours_of_cube(node*& box_ref, int& number);

void fast_LJ_update(point* r, point* dv, node**& box_ref, int N);

#endif
```

main_polymer.cc

```
#include<iostream>
#include<iomanip>
#include<math.h>
#include<fstream>
#include<stdlib.h>
#include<gsl/gsl_rng.h>
#include<gsl/gsl_randist.h>
#include<stdio.h>
#include"point.hh"
#include"ipoint.hh"
#include"ipoint_map.hh"
#include"constants.hh"
#include"potentials.hh"
#include"verlet.hh"
#include"config_utils.hh"

using namespace std;

char* run_info;
char* rm_run_info;
char* backup_ckpt;
char* rm_backup;
int   previous_killed;

void init_strings(char* checkpoint_name) {
  int length  = strlen(checkpoint_name);
  run_info    = new char[    length + 20];
  rm_run_info = new char[    length + 20];
  backup_ckpt = new char[2 * length + 20];
  rm_backup   = new char[    length + 20];
  sprintf(run_info, "%s.info", checkpoint_name);
  sprintf(rm_run_info, "rm -f %s", run_info);
  sprintf(backup_ckpt, "mv %s %s.bak", checkpoint_name, checkpoint_name);
  sprintf(rm_backup, "rm -f %s.bak", checkpoint_name);
  FILE* test_killed = fopen(run_info, "r");
  if (test_killed) {
    printf("Old info file found, restoring old state\n");
    previous_killed = 1;
    fclose(test_killed);
  } else {
    printf("No info file found, starting from new state\n");
    previous_killed = 0;
  }
}

void init_reread_restart_information(int& old_timesteps) {
  FILE *old_run_info = fopen(run_info, "r");                              // Test for previously killed process
  if (old_run_info) {
    int initial_old_timesteps;
    fscanf(old_run_info, "%i", &initial_old_timesteps);                   // read old process data
    if (old_timesteps < 0) {                                              // process was in throw-away-phase
      throw_away    = - old_timesteps;
      old_timesteps = initial_old_timesteps;
    } else {
      steps      = steps - (old_timesteps - initial_old_timesteps);       // compute remaining steps
      throw_away = 0;                                                     // do not preform init-steps
    }
  } else {                                                                // No previously running process killed
    if (throw_away && old_timesteps) {                                       // Do no initialisation with data
      cerr << "Throw away may only be applied with old timesteps == 0!\n"; // already computed
      exit(20);
    }
    old_run_info = fopen(run_info, "w");                                  // write data for new process
    fprintf(old_run_info, "%i\n", old_timesteps);
  }
  printf("Starting from timestep %i, %i throw away and %i regular remaining\n", old_timesteps, throw_away, steps);
  fclose(old_run_info);
}

int main(int argc, char* argv[]) {
  if ((argc != 6) && (argc != 7)) {
    std::cerr << "Invalid number of arguments - \n";
    std::cerr << "Calling sequence: <program name> constant-file start-config-file number-of-monomers ";
    std::cerr << "checkpoint-file stdout-file [append-config-file] \n";
    exit(1);
    }
  init_strings(argv[4]);
  init_constants(argv[1]);
  if (argc == 7)
    append_configs = 1;
  else
    append_configs = 0;
  if (display_constants)
    displayconstants(argv[5]);
  point *r;
  point *v;
  node  **box_ref;
  int number;
  int old_timesteps;
  map_ipoint idx_map;
  if (previous_killed)      // avoid checking of start- and checkpoint-file if previously killed
    init_polymer (r, v, box_ref, idx_map, number, argv[4], argv[4], argv[3], old_timesteps);
  else
    init_polymer (r, v, box_ref, idx_map, number, argv[2], argv[4], argv[3], old_timesteps);
  init_reread_restart_information(old_timesteps);
  for (int i = -throw_away; i < 0; i++) {
    if (use_verlet)
      verlet_timestep(r, v, box_ref, idx_map, number, i);
    else
      no_verlet_timestep(r, v, box_ref, idx_map, number, i);
    if (i % output_steps == 0) {                          // Checkpoint during long initialization
      system(backup_ckpt);
      write_config(r, v, 0, number, i, argv[4], "w");
```

main_polymer.cc

```
  }
}
FILE* stdout_file = fopen(argv[5], "a");
fprintf(stdout_file, "# N = %i\n", number);
fclose(stdout_file);
for (int i = 0; i <= steps; i++) {
  if (i % output_steps == 0) {
    // ------------------------------------------------------------------------------------------
    // --- Compute possible measurements ------------------------------------------------------
    // ------------------------------------------------------------------------------------------
    point center     = rouse_mode(r, number, 0);          // Compute center of mass
    point mode_1     = rouse_mode(r, number, 1);          // first rouse-mode
    point gyration   = gyration_radius(r, number);        // radius of gyration
    point end_to_end = r[number-1] - r[0];                // end-to-end vector
    point min_x      = r[0];                              // find minima/ maxima
    point max_x      = r[0];
    point min_y      = r[0];
    point max_y      = r[0];
    point min_z      = r[0];
    point max_z      = r[0];
    int   min_x_mon  = 0;
    int   min_y_mon  = 0;
    int   min_z_mon  = 0;
    int   max_x_mon  = 0;
    int   max_y_mon  = 0;
    int   max_z_mon  = 0;
    for (int j = 0; j < number; j++) {
      if (min_x.x_() > r[j].x_()) {
        min_x_mon = j;
        min_x     = r[j];
      }
      if (max_x.x_() < r[j].x_()) {
        max_x_mon = j;
        max_x     = r[j];
      }
      if (min_y.y_() > r[j].y_()) {
        min_y_mon = j;
        min_y     = r[j];
      }
      if (max_y.y_() < r[j].y_()) {
        max_y_mon = j;
        max_y     = r[j];
      }
      if (min_z.z_() > r[j].z_()) {
        min_z_mon = j;
        min_z     = r[j];
      }
      if (max_z.z_() < r[j].z_()) {
        max_z_mon = j;
        max_z     = r[j];
      }
    }
    double contour = 0;                                   // Compute contour-length
    double min_bond_sqr = 0;                              // Minimum bond-length
    double max_bond_sqr = 0;                              // Maximum bond-length
    if (number > 1) {
      min_bond_sqr = abs_sqr(r[1]-r[0]);
      max_bond_sqr = abs_sqr(r[1]-r[0]);
    }
    for (int j = 1; j < number; j++){
      contour += abs(r[j]-r[j-1]);
      if (abs_sqr(r[j]-r[j-1]) < min_bond_sqr)
        min_bond_sqr = abs_sqr(r[j]-r[j-1]);
      if (abs_sqr(r[j]-r[j-1]) > max_bond_sqr)
        max_bond_sqr = abs_sqr(r[j]-r[j-1]);
    }
    point mean_speed   = rouse_mode(v, number, 0);        // compute mean-speed
    point ang_momentum = angular_momentum(r, v, number);  // compute angular momentum;
    double energy      = energy_all(r, v, box_ref, number);      // compute energy
    double e_pot_int   = energy_pot_int(r, v, box_ref, number);  // compute energy_pot,internal
    double e_pot_ext   = energy_pot_ext(r, v, box_ref, number);  // compute energy_pot,external
    double e_kin       = energy_kin(r, v, box_ref, number);      // compute energy_kin
    double cos_theta   = theta(r, number);                // compute cos_theta
    double bond_correl = 0;
    if (number > 1)                                       // compute correlation of first
      bond_correl = (r[number-1]-r[0])*(r[1]-r[0]);       // bond with the end-to-end-vector
    // ------------------------------------------------------------------------------------------
    // --- Begin writing data -----------------------------------------------------------------
    // ------------------------------------------------------------------------------------------
    system(backup_ckpt);
    write_config(r, v, energy, number, i + old_timesteps, argv[4], "w");     // Write checkpoint
    FILE* stdout_file = fopen(argv[5], "a");
    fprintf(stdout_file, "%i %lf %lf %lf", i+old_timesteps, center.x_(), center.y_(), center.z_());
    if (display_mode_1)
      fprintf(stdout_file, "  %lf %lf %lf", mode_1.x_(), mode_1.y_(), mode_1.z_());
    if (display_gyration)
      fprintf(stdout_file, "  %lf %lf %lf", gyration.x_(), gyration.y_(), gyration.z_());
    if (display_end_to_end)
      fprintf(stdout_file, "  %lf %lf %lf", end_to_end.x_(), end_to_end.y_(), end_to_end.z_());
    if (display_min_max) {
      fprintf(stdout_file, "  %lf %lf %lf", min_x.x_(), min_y.y_(), min_z.z_());
      fprintf(stdout_file, "  %lf %lf %lf", max_x.x_(), max_y.y_(), max_z.z_());
    }
    if (display_min_max_mon) {
      fprintf(stdout_file, "  %i %i %i", min_x_mon, min_y_mon, min_z_mon);
      fprintf(stdout_file, "  %i %i %i", max_x_mon, max_y_mon, max_z_mon);
    }
    if (display_min_max_all) {
      fprintf(stdout_file, "  %i %lf %lf %lf", min_x_mon, min_x.x_(), min_x.y_(), min_x.z_());
      fprintf(stdout_file, "  %i %lf %lf %lf", min_y_mon, min_y.x_(), min_y.y_(), min_y.z_());
      fprintf(stdout_file, "  %i %lf %lf %lf", min_z_mon, min_z.x_(), min_z.y_(), min_z.z_());
      fprintf(stdout_file, "  %i %lf %lf %lf", max_x_mon, max_x.x_(), max_x.y_(), max_x.z_());
      fprintf(stdout_file, "  %i %lf %lf %lf", max_y_mon, max_y.x_(), max_y.y_(), max_y.z_());
```

main_polymer.cc

```
        fprintf(stdout_file, "  %i %lf %lf %lf", max_z_mon, max_z.x_(), max_z.y_(), max_z.z_());
      }
      if (display_contour)
        fprintf(stdout_file, "  %lf", contour);
      if (display_min_max_bond)
        fprintf(stdout_file, "  %lf %lf", sqrt(min_bond_sqr), sqrt(max_bond_sqr));
      if (display_mean_speed)
        fprintf(stdout_file, "  %lf %lf %lf", mean_speed.x_(), mean_speed.y_(), mean_speed.z_());
      if (display_ang_momentum)
        fprintf(stdout_file, "  %lf %lf %lf", ang_momentum.x_(), ang_momentum.y_(), ang_momentum.z_());
      if (display_energy)
        fprintf(stdout_file, "  %lf", energy);
      if (display_split_energy)
        fprintf(stdout_file, "  %lf %lf %lf", e_pot_int, e_pot_ext, e_kin);
      if (display_cos_theta)
        fprintf(stdout_file, "  %lf", cos_theta);
      if (display_bond_correl)
        fprintf(stdout_file, "  %lf", bond_correl);
      if (display_current_factor)
        fprintf(stdout_file, "  %lf", current_factor_el);
      fprintf(stdout_file, "\n");
      fclose(stdout_file);
      if (((i + old_timesteps) % config_steps == 0) && (append_configs) &&
          ((i != 0) || (throw_away != 0) || (old_timesteps == 0)))
        write_config(r, v, energy, number, i + old_timesteps, argv[6], "a");   // Append checkpoint, if desired
      // ------------------------------------------------------------------------------------------
      // --- End writing data -----------------------------------------------------------------
      // --------------------------------------------------------------------------------------
    }
    if (i != steps)                                                    // Compute next timestep
      if (use_verlet)
        verlet_timestep(r, v, box_ref, idx_map, number, i + old_timesteps);
      else
        no_verlet_timestep(r, v, box_ref, idx_map, number, i + old_timesteps);
  }
  write_config(r, v, energy_all(r, v, box_ref, number), number, steps + old_timesteps, argv[4], "w");
  system(rm_backup);
  system(rm_run_info);                                                  // Remove restart-information
}
```

makefile

```
CC=g++
CFLAGS=-O3 -I/usr/local/include
LDFLAGS=-L/usr/local/lib -lm -lgsl -lgslcblas

polymer : config_utils.o constants.o ipoint_map.o main_polymer.o potentials.o verlet.o
	$(CC) -o $@ $+ $(CFLAGS) $(LDFLAGS) -static

clean :
	rm -f *.o polymer

%.o: %.cc
	$(CC) -c $(CFLAGS) $<
```

point.hh

```
#include<iostream>
#include<iomanip>
#include<math.h>

#ifndef POINT_INCLUDED
#define POINT_INCLUDED

using namespace std;

// This is a class of 3-dimensional points with the common algebraic rules.
// The function abs_sqr calculates the sqare of the length of a point,
// it avoides taking the square-root.

class point{
  double x; double y; double z;
public:
  point()
  {}
  point(double xx, double yy, double zz)
  {
    x = xx; y = yy; z = zz;
  }
  friend double abs(point p){
    return sqrt(p.x*p.x + p.y*p.y + p.z*p.z);
  }
  friend double abs_sqr(point p){
    return (p.x*p.x + p.y*p.y + p.z*p.z);
  }
  friend ostream& operator<<(ostream& os, point p){
    return os << "(" << setw(9) << p.x << "," << setw(9) << p.y << ","
              << setw(9) << p.z << ")";
  }
  double component(int choice){                  // allows direct access to a single
    switch(choice) {                             // component, 1 .. 3
    case 0 : return sqrt(x*x+y*y+z*z);        // 0 returns the absolute of the point
    case 1 : return x;
    case 2 : return y;
    case 3 : return z;
    }
  }
  double x_(void){
    return x;
  }
  double y_(void){
    return y;
  }
  double z_(void){
    return z;
  }
  friend point  operator+(point p1, point p2){
    return point(p1.x + p2.x, p1.y + p2.y, p1.z + p2.z);
  }
  friend point  operator*(point p, double d){
    return point(p.x * d, p.y * d, p.z * d);
  }
  friend point  operator*(double d, point p){
    return p * d;
  }
  friend double operator*(point p1, point p2){
    return p1.x*p2.x + p1.y*p2.y + p1.z*p2.z;
  }
  friend point  operator-(point p){
    return (-1)*p;
  }
  friend point  operator-(point p1, point p2){
    return p1+(-p2);
  }
  friend point  operator/(point p, double d){
    return p*(1/d);
  }
  point  operator*=(double d){
    x *= d; y*=d; z*=d;
    return point(x,y,z);
  }
  point  operator+=(point p){
    x += p.x; y += p.y; z += p.z;
    return point(x,y,z);
  }
  point  operator-=(point p){
    x -= p.x; y -= p.y; z -= p.z;
    return point(x,y,z);
  }
  point  operator/=(double d){
    x /= d; y /= d; z /= d;
    return point(x,y,z);
  }
  friend bool   operator==(point p1, point p2){
    return (p1.x == p2.x) && (p1.y == p2.y) && (p1.z == p2.z);
  }
  friend bool   operator!=(point p1, point p2){
    return !(p1 == p2);
  }
  friend point vectorprod(point a, point b){
    return point(a.y*b.z - a.z*b.y, a.z*b.x - a.x*b.z, a.x*b.y - a.y*b.x);
  }
};

#endif
```

potentials.cc

```
#include<math.h>
#include<stdio.h>
#include<stdlib.h>
#include<gsl/gsl_rng.h>
#include<gsl/gsl_randist.h>
#include"point.hh"
#include"constants.hh"

#ifndef POTENTIALS_INCLUDED
#define POTENTIALS_INCLUDED

using namespace std;

// This file contains the interaction potentials for the monomers.
//
//  - The Lennard-Jones potential between two monomers, which is cut off at a
//    length of r = 2*sigma
//  - The bond-length potential between two bonds on the chain, limiting
//    the maximum bond-length
//  - The bond-angle-potential, which prefers bond-angle = 0
//  - The entropic-trap potential
//  - The spring-potential for the Rouse-model
//  - The entropic-trap potential
//  - The brownian motion-force
//  - Computation of the new scaling-factor for the pulsed external field
//  - The external electric field for the homogeneous case, the structured
//    microchannel, and the array device, without a potential

double sqr(double x) {
  return x*x;
}

double V_LJ(point r) {
  double r_2 = abs_sqr(r);
  if (r_2 > l_cutoff_LJ_sqr) return 0;
  double r_4 = r_2 * r_2;
  return epsilon_LJ*(sigma_6*sigma_6/(r_4*r_4*r_4) - sigma_6/(r_4*r_2))
         + v_cutoff_LJ;
}

point d_V_LJ_dr_times_timestep(point r) {
  double r_2 = abs_sqr(r);
  if (r_2 > l_cutoff_LJ_sqr) return point(0,0,0);
  double r_4 = r_2*r_2;
  double r_8 = r_4*r_4;
  return r * (prefactor_LJ * (1/r_8 - 2*sigma_6/(r_8*r_4*r_2)));
}

double V_BL(double d) {
  return -epsilon_BL/2 * d_BL_sqr *
         std::log(1 - (d-d_0_BL)*(d-d_0_BL) / d_BL_sqr);
}

point d_V_BL_dr_times_timestep(point r, double d) {
  double help = prefactor_BL / (1 - (d-d_0_BL)*(d-d_0_BL)/d_BL_sqr);
  return r * ((d-d_0_BL)/d * help);
}

double V_BA(point a, point b, double r_a, double r_b) {
  return epsilon_BA * (1 - a*b / (r_a*r_b));
}

point d_V_BA_da_times_timestep(point a, point b, double r_a, double r_b) {
  if (a == b) return point(0,0,0);
  point help = (a*b) * r_b/r_a * a - b * r_a * r_b;
  return help * (prefactor_BA / (r_a*r_a * r_b*r_b));
}

point d_V_BA_db_times_timestep(point a, point b, double r_a, double r_b) {
  if (a == b) return point(0,0,0);
  point help = (a*b) * r_a/r_b * b - a * r_a * r_b;
  return help * (prefactor_BA / (r_a*r_a * r_b*r_b));
}

double V_SP(point r) {
  return epsilon_SP/2 * abs_sqr(r);
}

point d_V_SP_dr_times_timestep(point r) {
  return prefactor_SP * r;
}

point d_V_BM_dr_times_delta_t(void) {
  static int init = 0;
  static gsl_rng* random = gsl_rng_alloc(gsl_rng_taus2);
  if (!init) {
    init = 1;
    gsl_rng_set(random, rand_seed);
    rand_seed = gsl_rng_get(random);
    if (save_new_seed) {
      FILE* rseed = fopen("random_seed", "w");
      fprintf(rseed, "%lu\n", rand_seed);
      fclose(rseed);
    }
  }
  double x = gsl_rng_uniform(random)-0.5;
  double y = gsl_rng_uniform(random)-0.5;
  double z = gsl_rng_uniform(random)-0.5;
  while (x*x + y*y + z*z > 0.25) {
    x = gsl_rng_uniform(random)-0.5;
    y = gsl_rng_uniform(random)-0.5;
    z = gsl_rng_uniform(random)-0.5;
  }
```

potentials.cc

```
  return prefactor_BM * point(x, y, z);
}

double V_LJ_wall(double r_2) {
  if (r_2 > l_cutoff_LJ_wall_sqr)
    return 0;
  double r_4 = r_2 * r_2;
  return epsilon_LJ_wall*(sigma_6_wall*sigma_6_wall/(r_4*r_4*r_4) - sigma_6_wall/(r_4*r_2))
         + v_cutoff_LJ_wall;
}

point d_V_LJ_wall_dr_times_timestep(point r) {
  double r_2 = abs_sqr(r);
  if (r_2 > l_cutoff_LJ_wall_sqr) return point(0,0,0);
  double r_4 = r_2*r_2;
  double r_8 = r_4*r_4;
  return r * (prefactor_LJ_wall * (1/r_8 - 2*sigma_6_wall/(r_8*r_4*r_2)));
}

double V_trap(point r) {
  double energy = 0;
  double x = r.x_();
  double z = r.z_();
  x -= std::floor(x / l_surface_trap) * l_surface_trap;
  if (z > (h_top_trap-l_cutoff_LJ_wall))
    energy += V_LJ_wall((z-h_top_trap)*(z-h_top_trap));
  if (z < l_cutoff_LJ_wall) {
    if (x > l_bottom_trap)
      energy += V_LJ_wall(z*z);
    else {
      if (z < l_cutoff_LJ_wall - h_bottom_trap)
        energy += V_LJ_wall((z + h_bottom_trap)*(z + h_bottom_trap));
      if (x < l_cutoff_LJ_wall + h_bottom_trap * wall_tilt_trap) {
        double z_ = 1 / (1 + wall_tilt_trap*wall_tilt_trap) * (z - wall_tilt_trap * x);
        if (z_ > 0)
          z_ = 0;
        double x_ = -wall_tilt_trap * z_;
        if ((x - x_) < l_cutoff_LJ_wall)
          energy += V_LJ_wall((x-x_)*(x-x_) + (z-z_)*(z-z_) );
      }
      if (x > l_bottom_trap - l_cutoff_LJ_wall - h_bottom_trap * wall_tilt_trap) {
        double z_ = 1 / (1 + wall_tilt_trap*wall_tilt_trap) * (z - wall_tilt_trap * (l_bottom_trap - x));
        if (z_ > 0)
          z_ = 0;
        double x_ = l_bottom_trap + wall_tilt_trap * z_;
        if ((x_ - x) < l_cutoff_LJ_wall)
          energy += V_LJ_wall((x-x_)*(x-x_) + (z-z_)*(z-z_));
      }
    }
  }
  return energy;
}

point d_V_trap_dr_times_timestep(point r) {
  point force = point(0,0,0);
  double x = r.x_();
  double z = r.z_();
  x -= std::floor(x / l_surface_trap) * l_surface_trap;
  if (z > (h_top_trap-l_cutoff_LJ_wall))
    force += d_V_LJ_wall_dr_times_timestep(point(0, 0, z-h_top_trap));
  if (z < l_cutoff_LJ_wall) {
    if (x > l_bottom_trap)
      force += d_V_LJ_wall_dr_times_timestep(point(0, 0, z));
    else {
      if (z < l_cutoff_LJ_wall - h_bottom_trap)
        force += d_V_LJ_wall_dr_times_timestep(point(0, 0, z + h_bottom_trap));
      if (x < l_cutoff_LJ_wall + h_bottom_trap * wall_tilt_trap) {
        double z_ = 1 / (1 + wall_tilt_trap*wall_tilt_trap) * (z - wall_tilt_trap * x);
        if (z_ > 0)
          z_ = 0;
        double x_ = -wall_tilt_trap * z_;
        if ((x - x_) < l_cutoff_LJ_wall)
          force += d_V_LJ_wall_dr_times_timestep(point(x-x_, 0, z-z_));
      }
      if (x > l_bottom_trap - l_cutoff_LJ_wall - h_bottom_trap * wall_tilt_trap) {
        double z_ = 1 / (1 + wall_tilt_trap*wall_tilt_trap) * (z - wall_tilt_trap * (l_bottom_trap - x));
        if (z_ > 0)
          z_ = 0;
        double x_ = l_bottom_trap + wall_tilt_trap * z_;
        if ((x_ - x) < l_cutoff_LJ_wall)
          force += d_V_LJ_wall_dr_times_timestep(point(x-x_, 0, z-z_));
      }
    }
  }
  return force;
}

double V_wall_y(point r) {
  double y = r.y_();
  return V_LJ_wall(y*y) + V_LJ_wall((y - wall_y)*(y - wall_y));
}

double V_mirror_trap(point r) {
  double potential = 0;
  double x = r.x_();
  double z = r.z_();
  x -= std::floor(x / l_surface_trap) * l_surface_trap;
  z -= std::floor(z / (2*h_total_trap)) * (2*h_total_trap);
  if (z <= l_cutoff_LJ_wall) {
    if (x <= l_cutoff_LJ_wall)
      potential += V_LJ_wall(x*x+z*z);
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      potential += V_LJ_wall((l_bottom_trap-x)*(l_bottom_trap-x)+z*z);
```

potentials.cc

```
    if (x >= l_bottom_trap)
      potential += V_LJ_wall(z*z);
  }
  if ((z >= 2*h_top_trap - l_cutoff_LJ_wall) && (z < 2*h_top_trap)) {
    if (x <= l_cutoff_LJ_wall)
      potential += V_LJ_wall(x*x+(2*h_top_trap-z)*(2*h_top_trap-z));
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      potential += V_LJ_wall((l_bottom_trap-x)*(l_bottom_trap-x)+(2*h_top_trap-z)*(2*h_top_trap-z));
    if (x >= l_bottom_trap)
      potential += V_LJ_wall((2*h_top_trap-z)*(2*h_top_trap-z));
  }
  if (z >= 2*h_top_trap) {
    if (x <= l_cutoff_LJ_wall)
      potential += V_LJ_wall(x*x);
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      potential += V_LJ_wall((l_bottom_trap-x)*(l_bottom_trap-x));
  }
  return potential;
}

point d_V_mirror_trap_dr_times_timestep(point r) {
  point force = point(0,0,0);
  double x = r.x_();
  double z = r.z_();
  x -= std::floor(x / l_surface_trap) * l_surface_trap;
  z -= std::floor(z / (2*h_total_trap)) * (2*h_total_trap);
  if (z <= l_cutoff_LJ_wall) {
    if (x <= l_cutoff_LJ_wall)
      force += d_V_LJ_wall_dr_times_timestep(point(x, 0, z));
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      force += d_V_LJ_wall_dr_times_timestep(point(x-l_bottom_trap, 0, z));
    if (x >= l_bottom_trap)
      force += d_V_LJ_wall_dr_times_timestep(point(0, 0, z));
  }
  if ((z >= 2*h_top_trap - l_cutoff_LJ_wall) && (z < 2*h_top_trap)) {
    if (x <= l_cutoff_LJ_wall)
      force += d_V_LJ_wall_dr_times_timestep(point(x, 0, z-2*h_top_trap));
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      force += d_V_LJ_wall_dr_times_timestep(point(x-l_bottom_trap, 0, z-2*h_top_trap));
    if (x >= l_bottom_trap)
      force += d_V_LJ_wall_dr_times_timestep(point(0, 0, z-2*h_top_trap));
  }
  if (z >= 2*h_top_trap) {
    if (x <= l_cutoff_LJ_wall)
      force += d_V_LJ_wall_dr_times_timestep(point(x, 0, 0));
    if ((x >= l_bottom_trap - l_cutoff_LJ_wall) && (x < l_bottom_trap))
      force += d_V_LJ_wall_dr_times_timestep(point(x-l_bottom_trap, 0, 0));
  }
  return force;
}

point d_V_wall_y_dr_times_timestep(point r) {
  double y = r.y_();
  return d_V_LJ_wall_dr_times_timestep(point(0, y, 0)) +
    d_V_LJ_wall_dr_times_timestep(point(0, y - wall_y, 0));
}

void set_current_factor(int timesteps){
  int reduced_timestep = timesteps % pulse_period_el;
  if (reduced_timestep < 0)                          // Correct timestep during initialization
    reduced_timestep += pulse_period_el;
  if (reduced_timestep < pulse_uptime_el)
    current_factor_el = 1;
  else
    current_factor_el = pulse_factor_el;
}

point d_V_el_dr_times_timestep(void) {
  return point(prefactor_el_x, prefactor_el_y, prefactor_el_z);
}

inline int idx(int i, int j) {
  return i * (trap_max_z+1) + j;
}

point d_V_el_trap_dr_times_timestep(point r){
  static int init = 0;
  static point (*field) = new point[(trap_max_x+1)*(trap_max_z+1)];
  if (!init) {
//-----------------------------------------------------------------------------
// begin initialisation
//-----------------------------------------------------------------------------
    double (*potential) = new double[(trap_max_x+1)*(trap_max_z+1)];
    double (*field_x)   = new double[(trap_max_x+1)*(trap_max_z+1)];
    double (*field_z)   = new double[(trap_max_x+1)*(trap_max_z+1)];
    for (int i = 0; i <= trap_max_x; i++)
      for (int j = 0; j <= trap_max_z; j++) {
        potential[idx(i,j)] = 0;
        field_x[idx(i,j)] = 0;
        field_z[idx(i,j)] = 0;
      }
    int number_of_points;
//-----------------------------------------------------------------------------
// read data from file
//-----------------------------------------------------------------------------
    FILE* input = fopen(potential_file_trap, "r");
    fscanf(input, "%*s %*i%*lf%*lf%*lf%*lf%*lf%*lf%*lf%i",&number_of_points);
    for(int i = 0; i < number_of_points; i++){
      double x, y, u;
      fscanf(input, "%le %le %le", &x, &y, &u);
      if (x <= 0)
        potential[idx(int(std::floor((x+l_surface_trap)/trap_discretize)+0.5),
          int(std::floor((y+h_bottom_trap)/trap_discretize)+0.5))]
```

potentials.cc

```
          = u*trap_scaling_factor + l_surface_trap*epsilon_el_trap;
      if (x >= 0)
        potential[idx(int(std::floor((x)/trap_discretize)+0.5),
          int(std::floor((y+h_bottom_trap)/trap_discretize)+0.5))]
          = u*trap_scaling_factor;
    }
    fclose(input);
//--------------------------------------------------------------------------------------------
// compute field in x-direction
//--------------------------------------------------------------------------------------------
    for (int j = 0; j <= trap_max_z; j++) {
      field_x[idx(0,j)] = (potential[idx(1,j)] - potential[idx(0,j)]) / trap_discretize;
      for (int i = 1; i < trap_max_x; i++)
        field_x[idx(i,j)] = (potential[idx(i+1,j)] - potential[idx(i-1,j)]) / (2 * trap_discretize);
      field_x[idx(trap_max_x,j)] = field_x[idx(0,j)];
    }
    for (int j = 0; j < trap_max_z - trap_top_z; j++) {
      field_x[idx(trap_max_x - trap_top_x,j)]
        = (potential[idx(trap_max_x - trap_top_x,j)] -
           potential[idx(trap_max_x - trap_top_x-1,j)]) / trap_discretize;
      for (int i = trap_max_x-trap_top_x+1; i < trap_max_x; i++)
        field_x[idx(i,j)] = 0;
    }
//--------------------------------------------------------------------------------------------
// compute field in z-direction
//--------------------------------------------------------------------------------------------
    for (int i = 0; i <= trap_max_x; i++) {
      field_z[idx(i,0)]          = (potential[idx(i,1)] - potential[idx(i,0)]) / trap_discretize;
      for (int j = 1; j < trap_max_z; j++)
        field_z[idx(i,j)] = (potential[idx(i,j+1)] - potential[idx(i,j-1)]) / (2 * trap_discretize);
      field_z[idx(i,trap_max_z)] = (potential[idx(i,trap_max_z)] - potential[idx(i,trap_max_z-1)]) /
        trap_discretize;
    }
    for (int i = trap_max_x-trap_top_x+1; i < trap_max_x; i++) {
      field_z[idx(i,trap_max_z-trap_top_z)] = (potential[idx(i,trap_max_z-trap_top_z+1)] -
                                               potential[idx(i,trap_max_z-trap_top_z)]) / trap_discretize;
      for (int j = 0; j < trap_max_z-trap_top_z; j++)
        field_z[idx(i,j)] = 0;
    }
//--------------------------------------------------------------------------------------------
// fill field-array with data
//--------------------------------------------------------------------------------------------
    double prefactor = delta_t;
    if (use_verlet)
      prefactor /= 2;
    for (int i = 0; i <= trap_max_x; i++)
      for (int j = 0; j <= trap_max_z; j++)
        field[idx(i,j)] = point(field_x[idx(i,j)],0,field_z[idx(i,j)]) * prefactor;
//--------------------------------------------------------------------------------------------
// delete temporay data
//--------------------------------------------------------------------------------------------
    delete []potential;
    delete []field_x;
    delete []field_z;
    init = 1;
//--------------------------------------------------------------------------------------------
// end initialisation
//--------------------------------------------------------------------------------------------
  }
  double x = r.x_();
  double z = r.z_();
  x -= std::floor(x / l_surface_trap) * l_surface_trap;
  z += h_bottom_trap;
  int i_x = int(std::floor(x / trap_discretize));
  int i_z = int(std::floor(z / trap_discretize));
  double lambda_x = x / trap_discretize - i_x;
  double lambda_z = z / trap_discretize - i_z;
  return lambda_z *(lambda_x*field[idx(i_x+1,i_z+1)] + (1-lambda_x)*field[idx(i_x,i_z+1)]) +
    (1-lambda_z)*(lambda_x*field[idx(i_x+1,i_z)] + (1-lambda_x)*field[idx(i_x,i_z)]);
}

point d_V_el_mirror_trap_dr_times_timestep(point r) {
  double x = r.x_();
  double z = r.z_();
// shift z-variable to unit cell (x will be shifted during computation of the field)
  z -= std::floor(z / (2*h_total_trap)) * (2*h_total_trap);
  if (z <= h_top_trap)
    return d_V_el_trap_dr_times_timestep(point(x, 0, z));                      // Usual trap
  if (z >= 2*h_top_trap + h_bottom_trap)
    return d_V_el_trap_dr_times_timestep(point(x, 0, z - 2*h_total_trap));     // Shifted, usual trap
  point force = d_V_el_trap_dr_times_timestep(point(x, 0, 2*h_top_trap - z));
  return point(force.x_(), 0, -force.z_());                                    // Mirrored force
}

#endif
```

potentials.hh

```
#include<math.h>
#include<stdio.h>
#include<gsl/gsl_rng.h>
#include<gsl/gsl_randist.h>
#include"point.hh"
#include"constants.hh"

#ifndef POTENTIALS_INCLUDED
#define POTENTIALS_INCLUDED

using namespace std;

// This file contains the interaction potentials for the monomers.
//
//  - The Lennard-Jones potential between two monomers, which is cut off at a
//    length of r = 2*sigma
//  - The bond-length potential between two bonds on the chain, limiting
//    the maximum bond-length
//  - The bond-angle-potential, which prefers bond-angle = 0
//  - The entropic-trap potential
//  - The spring-potential for the Rouse-model
//  - The brownian motion-force
//  - Computation of the new scaling-factor for the pulsed external field
//  - The external electric field, without a potential, for the homogeneous,
//    the structured microchannel, and the array device

double V_LJ(point r);

point d_V_LJ_dr_times_timestep(point r);

double V_BL(double d);

point d_V_BL_dr_times_timestep(point r, double d);

double V_BA(point a, point b, double r_a, double r_b);

point d_V_BA_da_times_timestep(point a, point b, double r_a, double r_b);
point d_V_BA_db_times_timestep(point a, point b, double r_a, double r_b);

double V_SP(point r);

point d_V_SP_dr_times_timestep(point r);

point d_V_BM_dr_times_delta_t(void);

double V_LJ_trap(double r_2);

double V_trap(point r);

point d_V_trap_dr_times_timestep(point r);

double V_mirror_trap(point r);

point d_V_mirror_trap_dr_times_timestep(point r);

double V_wall_y(point r);

point d_V_wall_y_dr_times_timestep(point r);

void set_current_factor(int timesteps);

point d_V_el_dr_times_timestep(void);

point d_V_el_mirror_trap_dr_times_timestep(point r);

point d_V_el_trap_dr_times_timestep(point r);

#endif
```

random_seed
1057223309

verlet.cc

```
#include<fstream>
#include<iostream>
#include<math.h>
#include<stdio.h>
#include"point.hh"
#include"ipoint.hh"
#include"ipoint_map.hh"
#include"constants.hh"
#include"potentials.hh"
#include"config_utils.hh"

#ifndef VERLET_INCLUDED
#define VERLET_INCLUDED

using namespace std;

// This file includes the Verlet-Algorithm, a function for initialisation of the polymer
// and a function for calculating the energy of the polymer.
//
// Functions used:
//  - init_polymer : reads the polymer-configuration from an input-file
//  - LJ-update    : computes the force on all monomers created by the Lennard-Jones-potential
//  - v-update     : computes all forces acting on all monomers created by the potentials,
//                   does not compute the brownian force or the friction force
//  - timestep     : this is the verlet-algorithm using all enabled forces (including
//                   friction and brownian force)
//  - r-update     : computes all forces acting on all monomers created by the potentials,
//                   does not compute the brownian force or the friction force
//  - no_verlet_timestep : this is the euler-algorithm using all enabled forces (including
//                   friction and brownian force) for overdamped dynamics
//  - energy_all     : computes the energy of the polymer, including all enabled potentials
//  - energy_pot_int : computes the potential energy of the polymer, regarding only internal potentials
//  - energy_pot_ext : computes the potential energy of the polymer, regarding only external potentials
//  - energy_kin     : computes the kinetic energy of the polymer

void init_polymer(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int& number, char* filename,
                  char* checkpoint_file, char* number_of_monomers, int& old_timesteps)
{
  number = atoi(number_of_monomers);
  FILE* input;
  if (valid_checkpoint(filename, checkpoint_file, number)) {
    printf("Using normal checkpoint '%s'\n", filename);
    input = fopen(filename, "r");
  } else {
    char* backup_name = new char[strlen(checkpoint_file)+10];
    sprintf(backup_name, "%s.bak", checkpoint_file);
    if (valid_checkpoint(backup_name, checkpoint_file, number)) {
      printf("Using backup '%s' and restoring checkpoint from backup\n", backup_name);
      input = fopen(backup_name, "r");
      char* command = new char[strlen(backup_name) + strlen(checkpoint_file) + 10];
      sprintf(command, "cp %s %s", backup_name, checkpoint_file);
      system(command);
    } else {
      cerr << "Neither startfile/checkpoint nor backup valid!\n";
      exit(5);
    }
    delete[] backup_name;
  }
  double x, y, z;
  int number_file;
  fscanf(input, "# %i # %i", &old_timesteps, &number_file);
  if (number > number_file) {                       // Make sure that there are enough monomers in
    cerr << "Too few monomers in start-file!\n";    // the start-file
    exit(6);
  }
  if ((strcmp(filename, checkpoint_file) == 0) && (number != number_file)) {      // When overwriting a checkpoint
    cerr << "Start- and checkpoint-file are identical and not all monomers used!\n"; // with itself , all monomers
    exit(7);                                                                       // have to be used
  }
  if (strcmp(filename, checkpoint_file) != 0){         // Do not overwrite a checkpoint if start-file is different
    FILE* test = fopen(checkpoint_file, "r");
    if (test) {
      cerr << "Start- and checkpoint-file differ and checkpoint-file exists!\n";
      exit(8);
    }
  }
  r = new point[number];
  v = new point[number];
  box_ref = new node*[number];
  fclose(input);                                    // Reset the read-buffer
  input = fopen(filename, "r");
  double trash;                                     // Start-files need not have a valid energy
  read_config(input, r, v, number, old_timesteps, trash, 1);
  for (int i = 0; i < number; i++)
    map_insert_index(idx_map, r[i]/l_cutoff_LJ, box_ref[i], i);
  fclose(input);
}

void v_update(point* r, point* v, point* dv, node**& box_ref, int N, int timesteps)
{
  double *d = new double[N];
  if (enable_BL || enable_BA)
    for (int i = 0; i <= N-2; i++) {
      d[i] = abs(r[i+1] - r[i]);
    }
  for (int i = 0; i < N; i++)
    dv[i] = point(0,0,0);
  if (enable_LJ)
    fast_LJ_update(r, dv, box_ref, N);
  if (enable_BL)
    for (int i = 0; i < N-1; i++) {
      if (d[i] >= d_0_BL + d_BL) {
        cerr << "Rupture at t = " << timesteps*delta_t << "(" << timesteps << " timesteps), monomer "
```

verlet.cc

```
            << i << "!\n";
        exit(14);
      }
      point V_BL_dr = d_V_BL_dr_times_timestep(r[i+1]-r[i], d[i]);
      dv[i]   +=  V_BL_dr;
      dv[i+1] += -V_BL_dr;
    }
  if (enable_BA)
    for (int i = 1; i < N-1; i++) {
      point V_BA_da = d_V_BA_da_times_timestep(r[i-1] - r[i], r[i] - r[i+1], d[i-1], d[i]);
      point V_BA_db = d_V_BA_db_times_timestep(r[i-1] - r[i], r[i] - r[i+1], d[i-1], d[i]);
      dv[i-1] += -V_BA_da;
      dv[i]   += (V_BA_da - V_BA_db);
      dv[i+1] +=  V_BA_db;
    }
  if (enable_SP)
    for (int i = 0; i < N-1; i++) {
      point d_V_SP = d_V_SP_dr_times_timestep(r[i+1]-r[i]);
      dv[i]   +=  d_V_SP;
      dv[i+1] += -d_V_SP;
    }
  if (enable_trap)
    if(enable_mirror_trap)
      for (int i = 0; i < N; i++)
        dv[i] += -d_V_mirror_trap_dr_times_timestep(r[i]);
    else
      for (int i = 0; i < N; i++)
        dv[i] += -d_V_trap_dr_times_timestep(r[i]);
  if (enable_wall_y)
    for (int i = 0; i < N; i++)
      dv[i] += -d_V_wall_y_dr_times_timestep(r[i]);
  if (enable_pulse_el)
    set_current_factor(timesteps);
  if (enable_el) {
    for (int i = 0; i < N; i++)
      dv[i] += -d_V_el_dr_times_timestep() * current_factor_el;
  }
  if (enable_el_trap)
    if (enable_mirror_trap)
      for (int i = 0; i < N; i++)
        dv[i] += -d_V_el_mirror_trap_dr_times_timestep(r[i]) * current_factor_el;
    else
      for (int i = 0; i < N; i++)
        dv[i] += -d_V_el_trap_dr_times_timestep(r[i]) * current_factor_el;
  delete []d;
}

void verlet_timestep(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int N, int timesteps) {
  static ipoint* old_cube;
  static point *dv;
  static int init = 0;
  if (!init) {
    init = 1;
    dv = new point[N];
    v_update(r, v, dv, box_ref, N, timesteps);
    old_cube = new ipoint[N];
    for (int i = 0; i < N; i++)
      old_cube[i] = ipoint(r[i]/l_cutoff_LJ);
  }
  for (int i = 0; i < N; i++) {
    if (enable_FR)
      v[i] *= decay_v;
    if (enable_BM)
      v[i] += d_V_BM_dr_times_delta_t();
    v[i] += dv[i];
  }
  for (int i = 0; i < N; i++) {
    r[i] += v[i] * delta_t;
    ipoint new_cube = ipoint(r[i]/l_cutoff_LJ);
    if (old_cube[i] != new_cube) {
      map_remove_index(idx_map, old_cube[i], box_ref[i], i);
      map_insert_index(idx_map, new_cube, box_ref[i], i);
      old_cube[i] = new_cube;
    }
  }
  v_update(r, v, dv, box_ref, N, timesteps);
  for (int i = 0; i < N; i++) {
    v[i] += dv[i];
  }
}

void r_update(point* r, point* dr, node**& box_ref, int N, int timesteps)
{
  static double *d = new double[N];
  if (enable_BL || enable_BA)
    for (int i = 0; i <= N-2; i++) {
      d[i] = abs(r[i+1] - r[i]);
    }
  for (int i = 0; i < N; i++)
    dr[i] = point(0,0,0);
  if (enable_LJ)
    fast_LJ_update(r, dr, box_ref, N);
  if (enable_BL)
    for (int i = 0; i < N-1; i++) {
      if (d[i] >= d_0_BL + d_BL) {
        cerr << "Rupture at t = " << timesteps*delta_t << "(" << timesteps << " timesteps), monomer "
             << i << "!\n";
        exit(15);
      }
      point V_BL_dr = d_V_BL_dr_times_timestep(r[i+1]-r[i], d[i]);
      dr[i]   +=  V_BL_dr;
      dr[i+1] += -V_BL_dr;
    }
```

verlet.cc

```
  if (enable_BA)
    for (int i = 1; i < N-1; i++) {
      point V_BA_da = d_V_BA_da_times_timestep(r[i-1] - r[i], r[i] - r[i+1], d[i-1], d[i]);
      point V_BA_db = d_V_BA_db_times_timestep(r[i-1] - r[i], r[i] - r[i+1], d[i-1], d[i]);
      dr[i-1] += -V_BA_da;
      dr[i]   += (V_BA_da - V_BA_db);
      dr[i+1] +=  V_BA_db;
    }
  if (enable_SP)
    for (int i = 0; i < N-1; i++) {
      point d_V_SP = d_V_SP_dr_times_timestep(r[i+1]-r[i]);
      dr[i]   +=  d_V_SP;
      dr[i+1] += -d_V_SP;
    }
  if (enable_trap)
    if(enable_mirror_trap)
      for (int i = 0; i < N; i++)
        dr[i] += -d_V_mirror_trap_dr_times_timestep(r[i]);
    else
      for (int i = 0; i < N; i++)
        dr[i] += -d_V_trap_dr_times_timestep(r[i]);
  if (enable_wall_y)
    for (int i = 0; i < N; i++)
      dr[i] += -d_V_wall_y_dr_times_timestep(r[i]);
  if (enable_pulse_el)
    set_current_factor(timesteps);
  if (enable_el)
    for (int i = 0; i < N; i++)
      dr[i] += -d_V_el_dr_times_timestep() * current_factor_el;
  if (enable_el_trap)
    if (enable_mirror_trap)
      for (int i = 0; i < N; i++)
        dr[i] += -d_V_el_mirror_trap_dr_times_timestep(r[i]) * current_factor_el;
    else
      for (int i = 0; i < N; i++)
        dr[i] += -d_V_el_trap_dr_times_timestep(r[i]) * current_factor_el;
}

void no_verlet_timestep(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int N, int timesteps) {
  static ipoint* old_cube;
  static point *dr;
  static int init = 0;
  if (!init) {
    init = 1;
    dr = new point[N];
    old_cube = new ipoint[N];
    for (int i = 0; i < N; i++)
      old_cube[i] = ipoint(r[i]/l_cutoff_LJ);
  }
  r_update(r, dr, box_ref, N, timesteps);
  if (enable_BM)
    for (int i = 0; i < N; i++)
      dr[i] += d_V_BM_dr_times_delta_t();
  for (int i = 0; i < N; i++) {
    r[i] += dr[i];
    v[i]  = dr[i] / delta_t;
    ipoint new_cube = ipoint(r[i]/l_cutoff_LJ);
    if (old_cube[i] != new_cube) {
      map_remove_index(idx_map, old_cube[i], box_ref[i], i);
      map_insert_index(idx_map, new_cube, box_ref[i], i);
      old_cube[i] = new_cube;
    }
  }
}

double energy_all(point* r, point* v, node**& box_ref, int number) {
  double energy = 0;
// ------------------------------------------------------------------------
  for (int k = 0; k<number; k++)              // kinetic energy
    energy += v[k]*v[k]/2;
// ------------------------------------------------------------------------
  if (enable_LJ)                              // Lennard-Jones
    for (int k = 0; k<number; k++) {
      int number;
      int* neighbours = neighbours_of_cube(box_ref[k], number);
      for (int j = 0; j < number; j++)
        if (neighbours[j] < k)
          energy += V_LJ(r[k]-r[neighbours[j]]);
      delete []neighbours;
    }
// ------------------------------------------------------------------------
  if (enable_BL)                              // bond-length
    for (int k = 0; k<number-1; k++)
      energy += V_BL(abs(r[k+1]-r[k]));
// ------------------------------------------------------------------------
  if (enable_BA)                              // bond-angle
    for (int k = 1; k<number-1; k++)
      energy+=V_BA(r[k-1]-r[k],r[k]-r[k+1],abs(r[k-1]-r[k]),abs(r[k+1]-r[k]));
// ------------------------------------------------------------------------
  if (enable_SP)                              // spring
    for (int k = 0; k<number-1; k++)
      energy += V_SP(r[k+1]-r[k]);
// ------------------------------------------------------------------------
  if (enable_trap)                            // entropic trap, mirrored
    if (enable_mirror_trap)
      for (int i = 0; i < number; i++)
        energy += V_mirror_trap(r[i]);
    else
      for (int i = 0; i < number; i++)        // pure entropic trap
        energy += V_trap(r[i]);
// ------------------------------------------------------------------------
  if (enable_wall_y)                          // wall_y
    for (int i = 0; i < number; i++)
```

verlet.cc

```
      energy += V_wall_y(r[i]);
// ------------------------------------------------------------------------
  return energy;
}

double energy_pot_int(point* r, point* v, node**& box_ref, int number) {
  double energy = 0;
// ------------------------------------------------------------------------
  if (enable_LJ)                              // Lennard-Jones
    for (int k = 0; k<number; k++) {
      int number;
      int* neighbours = neighbours_of_cube(box_ref[k], number);
      for (int j = 0; j < number; j++)
        if (neighbours[j] < k)
          energy += V_LJ(r[k]-r[neighbours[j]]);
      delete []neighbours;
    }
// ------------------------------------------------------------------------
  if (enable_BL)                              // bond-length
    for (int k = 0; k<number-1; k++)
      energy += V_BL(abs(r[k+1]-r[k]));
// ------------------------------------------------------------------------
  if (enable_BA)                              // bond-angle
    for (int k = 1; k<number-1; k++)
      energy+=V_BA(r[k-1]-r[k],r[k]-r[k+1],abs(r[k-1]-r[k]),abs(r[k+1]-r[k]));
// ------------------------------------------------------------------------
  if (enable_SP)                              // spring
    for (int k = 0; k<number-1; k++)
      energy += V_SP(r[k+1]-r[k]);
// ------------------------------------------------------------------------
  return energy;
}

double energy_pot_ext(point* r, point* v, node**& box_ref, int number) {
  double energy = 0;
// ------------------------------------------------------------------------
  if (enable_trap)                            // entropic trap, mirrored
    if (enable_mirror_trap)
      for (int i = 0; i < number; i++)
        energy += V_mirror_trap(r[i]);
    else
      for (int i = 0; i < number; i++)        // pure entropic trap
        energy += V_trap(r[i]);
// ------------------------------------------------------------------------
  if (enable_wall_y)                          // wall_y
    for (int i = 0; i < number; i++)
      energy += V_wall_y(r[i]);
// ------------------------------------------------------------------------
  return energy;
}

double energy_kin(point* r, point* v, node**& box_ref, int number) {
  double energy = 0;
// ------------------------------------------------------------------------
  for (int k = 0; k<number; k++)              // kinetic energy
    energy += v[k]*v[k]/2;
// ------------------------------------------------------------------------
  return energy;
}

#endif
```

verlet.hh

```
#include<fstream>
#include<iostream>
#include<math.h>
#include<stdio.h>
#include"point.hh"
#include"ipoint.hh"
#include"ipoint_map.hh"
#include"constants.hh"
#include"potentials.hh"

#ifndef VERLET_INCLUDED
#define VERLET_INCLUDED

using namespace std;

// This file includes the Verlet-Algorithm, a function for initialisation of the polymer
// and a function for calculating the energy of the polymer.
//
// Functions used:
//  - init_polymer   : reads the polymer-configuration from an input-file
//  - timestep       : this is the verlet-algorithm using all enabled forces (including
//                     friction and brownian force)
//  - no_verlet_timestep : this is the euler-algorithm using all enabled forces (including
//                     friction and brownian force) for overdamped dynamics
//  - energy_all     : computes the energy of the polymer, including all enabled potentials
//  - energy_pot_int : computes the potential energy of the polymer, regarding only internal potentials
//  - energy_pot_ext : computes the potential energy of the polymer, regarding only external potentials
//  - energy_kin     : computes the kinetic energy of the polymer

void init_polymer(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int& number, char* filename,
                  char* checkpoint_file, char* number_of_monomers, int& old_timesteps);

void verlet_timestep(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int N, int timesteps);
void no_verlet_timestep(point*& r, point*& v, node**& box_ref, map_ipoint& idx_map, int N, int timesteps);

double energy_all    (point* r, point* v, node**& box_ref, int number);
double energy_pot_int(point* r, point* v, node**& box_ref, int number);
double energy_pot_ext(point* r, point* v, node**& box_ref, int number);
double energy_kin    (point* r, point* v, node**& box_ref, int number);

#endif
```

List of Figures

Bibliography

[1] R. M. Dowben. *Cell Biology*. Harper & Row, 1971.

[2] S. Arnott, M. F. Wilkins, L. D. Hamilton, and R. Langridge. Fourier synthesis studies of lithium dna. 3. hoogsteen models. *J. Mol. Biol.*, 11:391–402, 1965.

[3] J. D. WATSON and F. H. CRICK. Molecular structure of nucleic acids; a structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[4] R. A. Bambara, R. S. Murante, and L. A. Henricksen. Enzymes and reactions at the eukaryotic dna replication fork. *Journal of Biological Chemistry*, 272(8):4647–4650, 1997.

[5] S. Waga and B. Stillman. The dna replication fork in eukaryotic cells. *Annual Review of Biochemistry*, 67:721–751, 1998.

[6] J.-L. Viovy. Electrophoresis of dna and other polyelectrolytes: Physical mechanisms. *Rev. Mod. Phys.*, 72:813–872, 2000.

[7] A. Grosberg. Statistical mechanics of protein folding: Some outstanding problems. In N. Attig, K. Binder, H. Grubmüller, and K. Kremer, editors, *Computational Soft Matter: From Synthetic Polymers to Proteins*, pages 375–400, Jülich, Germany, 2004. John von Neumann Institue for Computing (NIC).

[8] A. P. Gultyaev, F. H. D. Batenburg, and C. W. A. Pleij. The computer simulation of rna folding pathways using a genetic algorithm. *Journal of Molecular Biology*, 250(1):37–51, 1995.

[9] D. E. Smith, T. T. Perkins, and S. Chu. Self diffusion of an entangled dna molecule by reptation. *Phys. Rev. Lett.*, 75(22):4146–4149, 1995.

[10] M. N. Spiteri, F. Bouó, A. Lapp, and J. P. Cotton. Polyelectrolyte persistence length in semidilute solution as a function of the ionic strength. *Physica B*, 234-236:303–305, 1997.

[11] B. Tinland, A. Pluen, J. Sturm, and G. Weill. Persistence length of single-stranded dna. *Macromolecules*, 30:5763–5765, 1997.

[12] B. Dünweg. Advanced simulations for hydrodynamic problems: Lattice boltzmann and dissipative particle dynamics. In N. Attig, K. Binder, H. Grubmüller,

and K. Kremer, editors, *Computational Soft Matter: From Synthetic Polymers to Proteins*, pages 61–82, Jülich, Germany, 2004. John von Neumann Institue for Computing (NIC).

[13] G. W. Slater, C. Desruisseaux, S. J. Hubert, J.-F. Mercier, J. Labrie, J. Boileau, F. Tessier, and M. P. Pépin. Theory of dna electrophoresis: A look at some current challenges. *Electrophoresis*, 21:3873–3887, 2000.

[14] S. C. Jakeway, A. J. de Mello, and E. L. Russel. Miniaturized total analysis systems for biological analysis. *Fresenius J. Anal. Chem.*, 366:525–539, 2000.

[15] J. Krüger, K. Singh, A. O'Neill, C. Jackson, A. Morrison, and P. O'Brien. Development of a microfluidic device dor fluorescence activated cell sorting. *J. Micromech. Microeng.*, 12:486–494, 2002.

[16] C. S. Effenhauser, G. J. M. Bruin, and A. Paulus. Integrated chip-based capillary electrophoresis. *Electrophoresis*, 18:2203–2213, 1997.

[17] P. J. Shrewsbury, S. J. Muller, and D. Liepmann. Characterization of dna flow through microchannels. In *Technical Proceedings of the 1999 International Conference on Modeling and Simulation of Microsystems*, pages 578–580, 1999.

[18] R. M. Jendrejack, E. T. Dimalanta, D. C. Schwartz, M. D. Graham, and J. J. de Pablo. Dna dynamics in a microchannel. *Phys. Rev. Lett.*, 91(3):038102, 2003.

[19] R. Netz and D. Andelmann. Neutral and charged polymers at interfaces. *Phys. Rep.*, 380:1–95, 2003.

[20] T. Pfohl, J. H. Kim, M. Yasa, H. P. Miller, G. C. L. Wong, F. Bringezu, Z. Wen, L. Wilson, M. W.. Kim, Y. Li, and C. R. Safina. Controlled modification of microstructured silicon surfaces for confinement of biological macromolecules and liquid crystals. *Langmuir*, 17:5343–5351, 2001.

[21] P. J. Shrewsbury, S. J. Muller, and D. Liepmann. Flow of λ-dna in microfluidic devices. In *1st Annual International IEEE-EMBS Special Topic Conference on Microtechnologies in Medicine and Biology, Lyon, France, Oct. 12-14*, pages 415–420, 2000.

[22] D. Hennig and J. F. R. Archilla. Stretching and relaxation dynamics in double stranded dna. *Physica A*, 331:579–601, 2004.

[23] A. Ros, W. Hellmich, T. Duong, and D. Anselmetti. Towards single molecule analysis in pdms microdevices: from detection of ultra low dye concentrations to single dna molecule studies. *J. Biotechnology*, 112:65–72, 2004.

[24] N. Sundaram and D. K. Tafti. Evaluation of microchamber geometries and surface conditions for elektrokinetic driven mixing. *Anal. Chem.*, 76:3785–3793, 2004.

[25] J. B. Knight, A. Vishwanath, J. P. Brody, and R. H. Austin. Hydrodynamic focusing on a silicon chip: Mixing nanoliters in microseconds. *Phys. Rev. Lett.*, 80:3863–3866, 1998.

[26] C.-H. Lin, L.-M. Fu, and Y.-S. Chien. Microfluidic t-form mixer utilizing switching electroosmotic flow. *Anal. Chem.*, 76:5265–5272, 2004.

[27] H. A. Stone, A. D. Stock, and A. Ajdari. Engineering flows in small devices: Mocrofluidics towards lab-on-a-chip. *Annu. Rev. Fluid Mech.*, 36:381–411, 2004.

[28] S. Wiggings and J. Ottino. Foundations of chaotic mixing. *Phil. Trans. R. Soc. Lond. A*, 362:937–970, 2004.

[29] A. Han, L. Ceriotti, J. Lichtenberg, N. F. de Rooij, and E. Verpoorte. Dna fragmentation in a microfabricated microfluidic device. In *Proceedings of the μTAS 2003*, pages 575–587, 2003.

[30] P. G. DeGennes. Reptation of a polymer chain in the presence of fixed obstacles. *J. Chem. Phys.*, 55:572–579, 1971.

[31] L. S. Lerman and H. L. Frisch. Why does the electrophoretic mobility of dna in gels vary with the length of the molecule? *Biopolymers*, 21:995–997, 1982.

[32] O. J. Lumpkin and B. H. Zimm. Mobility of dna in gel electrophoresis. *Biopolymers*, 21:2315–2316, 1982.

[33] T. A. J. Duke, A. N. Semenov, and J. L. Viovy. Mobility of a reptating polymer. *Phys. Rev. Lett.*, 69:3260–3263, 1992.

[34] T. A. J. Duke, J. L. Viovy, and A. N. Semenov. Electrophoretic mobility of dna in gels. i. new biased reptation theory including fluctuations. *Biopolymers*, 34:239, 1994.

[35] O. Bakajin, T. A. J. Duke, J. Tegenfeldt, C.-F. Chou, S. S. Chan, R. H. Austin, and E. C. Cox. Separation of 100-kilobase dna molecules in 10 seconds. *Anal. Chem.*, 73:6053–6056, 2001.

[36] M. Burmeister and L. Ulanovsky. *Pulsed-Field Gel Electrophoresis: Protocols, Methods, and Theories*. Humana Press, Totowa, NJ, 1992.

[37] D. C. Schwartz and C. R. Cantor. Separation of yeast chromosome-sized dnas by pulsed field gradient gel electrophoresis. *Cell*, 37:67–75, 1984.

[38] M. J. Orbach, D. Vollrath, R. W. Davis, and C. Yanofsky. An electrophoretic karyotype of neurospora crassa. *Mol. Cell. Biol.*, 8(4):1469–1473, 1988.

[39] E. C. Cox, C. D. Vocke, S. Walter, K. Y. Gregg, and E. S. Bain. Electrophoretic karyotype for dictyostelium discoideum. *Proc. Natl. Sci. U.S.A.*, 87:8247–8251, 1990.

[40] L. R. Huang, J. O. Tegenfeldt, J. J. Kraeft, J. C. Sturm, R. H. Austin, and E. C. Cox. A dna prism for high-speed continuous fractionation of large dna molecules. *Nature Biotechnology*, 20:1048–1051, 2002.

[41] M. N. Albarghouthi and A. E. Barron. Polymeric matrices for dna sequencing by capillary electrophoresis. *Electrophoresis*, 21:4096–4111, 2000.

[42] P. D. Grossmann and J. C. Colburn. *Capillary Electrophoresis, Theory and Practice: Free Solution Capillary Electrophoresis.* Acad. Press, San Diego, 1992.

[43] P. G. Righetti. *Capillary Electrophoresis in Analytical Biotechnology.* CRC Series in Analytical Biotechnology (CRC, Boca Raton), 1996.

[44] C. Heller. *Analysis of Nucleic Acids by Capillary Electrophoresis.* Friedr. Vieweg & Sohn Verlagsgesellschaft, Braunschweig/ Weinheim, Germany, 1997.

[45] E. V. Dose and G. Guiochon. Timescales of transient processes in capillary electrophoresis. *Journal of Chromatography A*, 652:263–275, 1993.

[46] R. A. Mosher, C.-X. Zhang, J. Caslavska, and W. Thormann. Dynamic simulator for capillary electrophoresis with in situ calculation of electroosmosis. *J. Chromatography A*, 716:17–26, 1995.

[47] E. Marshall. Rival genome sequencers celebrate a milestone together. *Science*, 288:2304–2305, 2000.

[48] Elizabeth Pennisi. Finally, the book of life and instructions for navigating it. *Science*, 288:2304–2307, 2000.

[49] A. Baumgärtner and M. Muthukumar. A trapped polymer chain in random porous media. *J. Chem. Phys.*, 87(5):3082–3088, 1987.

[50] M. Muthukumar and A. Baumgärtner. Effects of entropic barriers on polymer dynamics. *Macromolecules*, 22:1937 – 1941, 1989.

[51] M. Muthukumar and A. Baumgärtner. Diffusion of a polymer chain in random media. *Macromolecules*, 22:1941 – 1946, 1989.

[52] G. W. Slater and S. Y. Wu. Reptation, entropic trapping, percolation, and rouse dynamics of polymers in "random" environments. *Phys. Rev. Lett.*, 75:164–167, 1995.

[53] J. Han and H. G. Craighead. Entropic trapping and sieving of long dna molecules in a nonafluidic channel. *J. Vac. Sci. Technol. A*, 17:2142–2147, 1998.

[54] J. Han, S. W. Turner, and H. G. Craighead. Entropic trapping and escape of long dna molecules at submicron size constriction. *Phys. Rev. Lett.*, 83:1688–1691, 1999. Erratum, Phys. Rev. Lett. **86**, 1394 (2001).

[55] J. Han and H. G. Craighead. Separation of long dna molecules in a microfabricated entropic trap array. *Science*, 288:1026–1029, 2000.

[56] J. Han and H. G. Craighead. Characterization and optimization of an entropic trap for dna separation. *Anal. Chem.*, 74:394–401, 2002.

[57] F. Tessier, J. Labrie, and G. W. Slater. Electrophoretic separation of long polyelectrolytes in submolecular-size constrictions: A monte carlo study. *Macromolecules*, 35:4791–4800, 2002.

[58] M. Streek, F. Schmid, T. T. Duong, and A. Ros. Mechanisms of dna separation in entropic trap arrays: A brownian dynamics simulation. *J. Biotechnology*, 112:79–89, 2004.

[59] Z. Chen and F. A. Escobedo. Simulation of chain-length partitioning in a microfabricated channel via entropic trapping. *Mol. Sim.*, 29:417–425, 2003.

[60] F. Tessier and G. W. Slater. Strategies for the separation of polyelectrolytes based on non-linear dynamics and entropic ratchets in a simple microfuidic device. *Appl. Phys. A*, 75:285–291, 2002.

[61] T. T. Duong. Dna-migration in strukturierten mikrofluidik-kanälen. diploma thesis, Universität Bielefeld, 2002.

[62] T. T. Duong, R. Ros, M. Streek, F. Schmid, J. Brugger, D. Anselmetti, and A. Ros. Size-dependent free solution dna electrophoresis in structured microfluidic systems. *Microelectronic Engineering*, 67-68:905–912, 2003.

[63] M. Streek, F. Schmid, T. T. Duong, D. Anselmetti, and A. Ros. Two-state migration of dna in a structured microchannel. *Phys. Rev. E*, page accepted, 2004.

[64] T. T. Duong, M. Streek, R. Ros, F. Schmid, A. Ros, and D. Anselmetti. Gel-free electrophoresis of λ- and T2-dna in structured pdms microfluidic devices. In *Proceedings of the μTAS 2003*, pages 749–752, 2003.

[65] G. I. Nixon and G. W. Slater. Entropic trapping and electrophoretic drift of a polyelectrolyte down a channel with a periodically oscillating width. *Phys. Rev. E*, 53:4969–4980, 1996.

[66] G. W. Slater, H. L. Guo, and G. I. Nixon. Bidirectional transport of polyelectrolytes using self-modulating entropic ratchets. *Phys. Rev. Lett.*, 78(6):1170–1173, 1997.

[67] T. Pfohl and S. Herminghaus. Mikrofluidik mit komplexen flüssigkeiten. *Physik Journal*, 2(1):35–40, 2003.

[68] D. E. Smith, H. P. Babcock, and S. Chu. Single-polymer dynamics in steady shear flow. *Science*, 283:1724–1727, 1999.

[69] D. Schmalzing, L. Koutny, A. Adourian, P. Belgrader, P. Matsudaira, and D. Ehrlich. Dna typing in thirty seconds with a microfabricated device. *Proc. Nat. Ac. Sci: USA*, 94:10273–10278, 1997.

[70] M. Ueda, T. Hayama, Y. Takamura, Y. Horiike, and Y. Baba. Investigation of the possibility of geometrical electrophoresis. *Electrophoresis*, 23:2635–2641, 2002.

[71] H.-P. Chou, C. Spence, A. Scherer, and S. Quake. A microfabricated device for sizing and sorting dna molecules. *Proc. Nat. Ac. Sci: USA*, 96:11–13, 1999.

[72] S. W. Turner, A. M. Perez, A. Lopez, and H. G. Craighead. Monolithic nanofluidic sieving structures for dna manipulation. *J. Vac. Sci. Technol. B*, 16:3835–3840, 1998.

[73] H. Linke. Von dämonen und elektronen. *Physikalische Blätter*, 56:45–47, 2000.

[74] J. S. Bader, R. W. Hammond, S. A. Henck, M. W. Deem, G. A. McDermott, J. M. Bustillo, J. W. Simpson, G. T. Mulhern, and J. M. Rothberg. Dna transport by a micromachined brownian ratchet device. *Proc. Nat. Ac. Sci: USA*, 96:13165–13169, 1999.

[75] J. S. Bader, M. W. Deem, R. W. Hammond, S. A. Henck, J. W. Simpson, and J. M. Rothberg. A brownian-ratchet dna pump with applications to single-nucleotide polymorphism genotyping. *Appl. Phys. A*, 75:275–278, 2002.

[76] R. D. Astumian. Thermodynamics and kinetics of a brownian motor. *Science*, 276:917–922, 1997.

[77] C. Desruisseaux, G. W. Slater, and T. B. L. Kist. Trapping electrophoresis and ratchets: A theoretical study for dna-protein complexes. *Biophys. Journal*, 75:1228–1236, 1998.

[78] C. Marquet, A. Buguin, L. Talini, and P. Silberzahn. Rectified motion of colloids in asymmetrically structured channels. *Phys. Rev. Lett.*, 88:168301, 2002.

[79] Y. Takamura, Y. Horiike, Y. Baba, and E. Tamiya. Size dependent mobility of dna in electric and hydro drag force field. In *Proceedings of the μTAS 2003*, pages 1183–1186, 2003.

[80] D. Frenkel and B. Smit. *Understanding molecular Simulation: From Algorithms to Applications*. Academic Press, 1996.

[81] M. P. Allen. Introduction to molecular dynamics simulation. In N. Attig, K. Binder, H. Grubmüller, and K. Kremer, editors, *Computational Soft Matter: From Synthetic Polymers to Proteins*, pages 1–27, Jülich, Germany, 2004. John von Neumann Institue for Computing (NIC).

[82] M. Doi. *Introduction to Polymer Physics*. Oxford Science Publications, 1997. reprint1997.

[83] H. Risken. *The Fokker-Planck equation: methods of solution and applications.* Springer Verlag, Berlin, 1989.

[84] M. Doi and S. F. Edwards. *The Theory of Polymer Dynamics.* Oxford Science Publications, New York, 1986.

[85] B. Dünweg and W. Paul. Brownian dynamics simulation without gaussian random numbers. *Int. J. Mod. Phys. C*, 2:817–827, 1991.

[86] M. Streek. Migration of dna on a structured surface in an external field. diploma thesis, Universität Bielefeld, 2002.

[87] P. Reimann. Brownian motors: Noisy transport far from equilibrium. *Phys. Rep.*, 361:57–265, 2002.

[88] B. Dünweg and K. Kremer. Microscopic verification of dynamical scaling in dilute polymer solutions: A molecular-dynamics simulation. *Phys. Rev. Lett.*, 66:2996–2999, 1991.

[89] A. Kopf, B. Dünweg, and W. Paul. Dynamics of polymer "isotope" mixtures: Molecular dynamics simulation and rouse model analysis. *J. Chem. Phys.*, 107:6945–6955, 1997.

[90] J. C. LeGuillou and J. Zinn-Justin. Critical exponents for the n-vector model in three dimensions from field theory. *Phys. Rev. Lett.*, 39:95–98, 1977.

[91] D. S. McKenzie. Polymers and scaling. *Phys. Rep.*, 27c:35–88, 1976.

[92] J. M. Deutsch. Dynamics of pulsed-field electrophoresis. *Phys. Rev. Lett.*, 59:1255–1258, 1987.

[93] J. M. Deutsch. Theoretical studies of dna during gel electrophoresis. *Science*, 240:922–924, 1988.

[94] J. M. Deutsch and T. L. Madden. Theoretical studies of dna during gel electrophoresis. *J. Chem. Phys.*, 90:2476–2485, 1989.

[95] M. Matsumoto and M. Doi. Brownian dynamics simulation of dna gel electrophoresis. *Mol. Sim.*, 12:219–226, 1994.

[96] H. Noguchi and M. Takasu. Dynamics of dna in entangled polymer solutions: An anisotropic friction model. *J. Chem. Phys.*, 114:7260–7266, 2001.

[97] K. Kremer. Entangled polymers: From universal aspects to structure property relations. In N. Attig, K. Binder, H. Grubmüller, and K. Kremer, editors, *Computational Soft Matter: From Synthetic Polymers to Proteins*, pages 1–27, Jülich, Germany, 2004. John von Neumann Institue for Computing (NIC).

[98] D. Ceperley, M. H. Karlos, and J. L. Lebowitz. Computer simulation of the dynamics of a single polymer chain. *Phys. Rev. Lett.*, 41:313–316, 1978.

[99] E. B. Cummings, S. K. Griffiths, R. H. Nilson, and P. H. Paul. Conditions for similitude between the fluid velocity and electric field in electroosmotic flow. *Anal. Chem.*, 72:2526–2532, 2000.

[100] J. G. Santiago. Electroosmotic flows in microchannels with finite inertial and pressure forces. *Anal. Chem.*, 73:2353–2365, 2001.

[101] D. Long, J.-L. Viovy, and A. Ajdari. Simultaneous action of electric fields and nonelectric forces on a polyelectrolye: motion and deformation. *Phys. Rev. Lett.*, 76:3858–3861, 1996.

[102] E. Stellwagen, Y. Lu, and N. C. Stellwagen. Unified description of electrophoresis and diffusion for dna and other polyions. *Biochemistry*, 42:11745–11750, 2003.

[103] P. G. DeGennes. *Scaling Concepts in Polymer Physics*. Cornel University Press, Ithaca,NY, 1979.

[104] A. Baumgärtner. Polymer scaling. In Jan K. G. Dhont, Gerhard Gompper, and Dieter Richter, editors, *Soft Matter - Complex Materials on Mesoscopic Scales*, page B3. 33. IFF-Ferienkurs, Forschungszentrum Jülich, 2002.

[105] E. Stellwagen and N. C. Stellwagen. The free solution mobility of dna in tris-acetate-edta buffers of different concentration, with and without added nacl. *Electrophoresis*, 23:1935–1941, 2002.

[106] D. E. Smith and S. Chu. Response of flexible polymers to a sudden elongational flow. *Science*, 281:1335–1339, 1998.

[107] A. S. Panwar and S. Kumar. Brownian dynamics simulations of polymer stretching and transport in a complex electroosmotic flow. *J. Chem. Phys.*, 118(2):925–936, 2003.

[108] L. R. Huang, J. O. Tegenfeldt, , J. J. Kraeft, J. C. Sturm, R. H. Austin, and E. C. Cox. Generation of large-area tunable uniform electric fields in microfluidic arrays for rapid dna separation. *Technical Digest of International Electron Devices Meeting*, pages 363–366, 2001.

[109] L. R. Huang, P. Silberzan, J. O. Tegenfeldt, E. C. Cox, J. C. Sturm, R. H. Austin, and H. Craighead. Role of molecular size in ratchet fractionation. *Phys. Rev. Lett.*, 89:178301, 2002.

[110] L. R. Huang, E. C. Cox, R. H. Austin, and J. C. Sturm. Tilted brownian ratchet for dna analysis. *Anal. Chem.*, 75:6963–6967, 2003.

[111] D. Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley, 1993.

[112] 2003. The Condor team. Software package from www.cs.wisc.edu/condor/.

Acknowledgements

At this point, I would like to express my gratitude to a few people and organizations who effectively supported this project/ An dieser Stelle möchte ich einigen Personen und Organisationen danken, die diese Arbeit nachhaltig unterstützt haben.

Prof. Dr. Friederike Schmid, die, auch in Zeiten schwerer Krankheit, stets Zeit und ein offenes Ohr für meine Fragen und Probleme hatte. Des Weiteren möchte ich mich bei ihr für einen Teil der Auswertung der Simulationsdaten zu den entropischen Fallen bedanken.

PD Dr. Robert Ros, der sich bereit erklärt hat, diese Arbeit zu begutachten und in sehr kurzer Zeit ein Gutachten zu erstellen.

Thanh Tu Duong, der mir bei Problemen stets hilfreich zur Seite stand, und meine Simulationen durch Experimente unterstützt hat. Seine aktive Zusammenarbeit hat sichergestellt, daß dieses Projekt so erfolgreich verlaufen ist.

Dr. Alexandra Ros, die die ursprüngliche Idee für dieses Projekt hatte. Durch ihren persönlichen Einsatz wurde das Projekt entscheidend vorangebracht, sowohl während der Antragstellung als auch auf experimenteller Seite.

Der Arbeitsgruppe der kondensierten Materie der Professoren Schmid und Reimann an der Universität Bielefeld für die Unterstützung meiner Arbeit und die vielen, anregenden Diskusionen.

Der Arbeitsgruppe "Experimentelle Biophysik und angewandte Nanowissenschaften" von Professor Anselmetti an der Universität Bielefeld, die ich durch die Zusammenarbeit mit Thanh Tu Duong und Dr. Alexandra Ros näher kennen gelernt habe.

Diese Arbeit ist als Teil des Teilprojekts D2 des Sonderforschungsbereichs 613 "Physik von Einzelmolekülprozessen und molekularer Erkennung", der von der DFG finanziert wurde, entstanden.

Parts of the results presented here were obtained on compute clusters using the job queuing system "Condor." The Condor Software Program (Condor) was developed by the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison. All rights, title, and interest in Condor are owned by the Condor Team [112].

Meinen Eltern, die mich während meines gesamten Studiums stets mit Rat und Tat unterstützt haben.