Bachelor thesis

Simulation of artificial and biological nanoparticles’ trajectories in hybrid force-fields

Submitted by

Simon Welker
Student number 6927991

in the study program
B. Sc. Computing in Science SP Chemie
at the Department of Informatics
of the MIN Faculty
of Universität Hamburg

October 2019

1. Examiner: Prof. Dr. Jochen Küpper
2. Examiner: Dr. Muhamed Amin
## Contents

1. Introduction ............................................. 1

2. Fundamental concepts .................................... 5
   2.1. Single-particle imaging ............................... 5
   2.2. Multiphysics trajectory simulations ................. 6
   2.3. Aerodynamical control of particle beams ............. 8
   2.4. Optical control of particle beams .................... 10

3. CMInject: An extensible framework for trajectory simulations 13
   3.1. Overview ........................................... 14
   3.2. Object class definitions ............................ 15
   3.3. Defining and running virtual experiment setups ........ 17
   3.4. Implementation of particle propagation ............... 17
   3.5. Output storage, analysis and visualization .......... 18

4. Simulations and results .................................. 21
   4.1. Result analysis of toy problem simulations .......... 21
       4.1.1. Radially symmetrical flow simulations in 3D and 2D . 21
       4.1.2. Asymmetrical 3D flow simulations in low temperatures . 22
   4.2. Comparisons against existing programs ............... 26
       4.2.1. The previous version of CMInject ................ 26
       4.2.2. particleTracing.py: A team-internal tool ......... 26
   4.3. Photophoretic focusing ................................ 28
   4.4. Multiphysics experiment setups ...................... 29

5. Performance evaluation .................................. 33
   5.1. Runtime performance overview ....................... 33
   5.2. Multiprocessing overhead ............................ 34
   5.3. Comparisons against existing programs ............... 36
       5.3.1. The previous version of CMInject ................ 37
       5.3.2. particleTracing.py: A team-internal tool ........ 38
   5.4. Profiler output for CMInject and particleTracing.py . 41
I’m very grateful to Muhamed and Jochen for allowing me to work with them on this multifaceted problem which was (luckily!) rather far outside my comfort zone and for their useful feedback, to Andrei Rode for his valuable input and always friendly manner, to Nils Roth for the informative and funny whiteboard sessions and discussions, to Lena Worbs for providing me with a detailed drawing and explanation of her experiment setup, to Amit Samanta for always being encouraging, to everyone in the Controlled Molecule Imaging group for being very pleasant and helpful people, to my parents for all the financial and emotional support that allowed me to complete this Bachelor’s degree, to my siblings and our family pets for always making home a place to laugh and relax, to Karim for all the integral bois, and to Theresa for supporting me all the way through.
Humans are a visual species. Ever since the existence of atoms has been confirmed without a doubt, scientists have been wishing to be able to see objects like atoms and molecules moving and interacting with each other. Chemists wish to unravel molecular structures and interactions to better understand the reactions between them, biologists want to be able to see and understand the complex machinery inside living beings in action, and nanoscientists depend on a way to image the structures they investigate and design. As opposed to seeing macroscale objects, imaging matter at the atomic scale is far from easy. Since atoms and even complex molecules like proteins are tiny and—under normal conditions—move about very quickly on scales much larger than themselves, hitting them with a sufficient number of photons that are then also reflected back in a way that creates a sharp image is a hard problem. Ways to image micro- and nanoscale objects have nonetheless been developed and improved in recent decades based on complex physical interactions and precise measurement devices. While they are yet far from what is generally thought of to be “seeing something”, these techniques have enabled disciplines like chemistry and biology to take a huge step forward. Among others, there are electron microscopy (EM) [1], nuclear magnetic resonance (NMR) spectroscopy [2], rotationally resolved spectroscopy [3,4], and x-ray imaging [5].

X-ray crystallography has been especially successful at resolving the structures of proteins with a resolution generally of a quality that allows predictions about structure-based interactions to be made. As such, the data retrieved by this method makes up the largest part of proteins’ structural data that has been collected. In this method, x-ray photons illuminate
1. Introduction

a protein crystal and diffract through interactions with electrons present in the proteins. The scattered photons yield a diffraction pattern that can be measured. Because these interactions follow certain physical laws, they exhibit certain patterns and one can reconstruct an electron density from a large enough amount of diffraction images. For proteins, one can then fit—at a computer—each single atom that makes up the protein into this measured electron density, until a plausible structure is created.

The glaring issue with x-ray crystallography is in its name: it depends on a crystallized sample having previously been prepared to be imaged. Many proteins are however difficult or even impossible to crystallize, so they and their structures elude this method. Furthermore, it is generally unlikely that a protein’s structure will be the same within a crystal and within the biological system it came from, so the imaged structures might well not tell the true picture. Only within the last decade, single-particle imaging (SPI) using x-rays has emerged as an alternative that does not have this limitation. Therein, a beam of many single particles is injected into a region where they interact with a certain probability with an x-ray laser. This interaction also creates diffraction patterns that are then recorded, and molecular structures can be reconstructed from them. The advantages of this method, if developed to its full potential, are manifold. First, as already mentioned, it does not depend on a crystallized sample; second, imaging single particles could circumvent the problem of a structural mismatch of the measured molecule and the molecule under normal conditions; and third, when the conditions are right, one could even reconstruct the movements of atoms inside these molecular particles, or the movements of particle parts as they interact with each other.

There are two principal requirements for successfully implementing the SPI method. The first are a pulsed x-ray laser source that is very bright and has a very short pulse duration. The brightness is required for collecting high-quality diffraction patterns since there must be a large number of photon-atom interactions. The short pulse length is necessary to balance out the destructive nature of high-intensity x-rays. It has been found that when using pulse lengths on the scale of femtoseconds ($10^{-15}$ s), one can expose samples to x-ray radiation doses of at least 1000 MGy while still collecting diffraction patterns seemingly coming from undisturbed molecules [6]. The reason this is possible is that atoms start moving much slower than it takes a femtosecond photon pulse to terminate [7]. Such sources, so-called free-electron lasers (FELs), have been built only within the last decade, with the European XFEL starting lasing in 2017.

One of the principal issues currently holding back x-ray single particle
imaging from becoming a feasible method to produce high-quality structures quickly is the fact that it is hard to control the motions of nanoparticles. As the focus size of the laser should be chosen to be as small as possible to provide maximum irradiance and is on the scale of nm to few µm, the probability of a single particle in flight interacting with the laser is low when it is not controlled to follow a path that will intersect this region. As the laser is pulsed and not continuous, with the European XFEL having a repetition rate of 10 Hz with 2700 laser pulses per repetition, particle velocities must also be controlled to increase the probability of hitting a particle with a beam. A particle ideally positioned to be imaged would be almost stationary directly within the laser’s interaction region and shockfrozen in its native structure. Achieving a beam of many of these particles is essential to be able to quickly and accurately resolve particle structures with SPI using x-ray free electron lasers, and it is an active area of research. Further experimental challenges in SPI are the design of high-quality detectors and the avoidance of background noise, which must also be addressed.

In almost all physics research, simulations are indispensable tools to accompany and enhance experiments and test out theoretical models in the computer. For the focusing of nanoparticle beams, simulations of particle moving on paths through space—their trajectories—as they are influenced by forces are useful to be able to analyze the effectiveness of different strategies involving varying experiment geometries, employed forces, and initial particle conditions before implementing them as experiments. The advantages of this are both in cost and time: Designing and realizing a real-world experiment and testing out different parameters generally both takes orders of magnitude longer and is orders of magnitude more expensive than defining and simulating an approximate computational model.

To enable physicists to easily define and run such nanoparticle trajectory simulations and analyze the results, an extensible framework for numerical nanoparticle trajectory simulations called “CMInject” was designed and implemented for this thesis. Based on a previous non-extensible single-purpose version, its structure was analyzed and redesigned from scratch. Full documentation and coding style best practices were introduced. Simulation results were compared against existing programs and against experiment. Performance analyses were carried out, comparing runtime and profiler output with existing programs, and performance was improved iteratively until program runtime was within a factor of 2 of an optimized single-purpose Python script. After a description of the fundamental concepts involved, the redesigned structure as well as the results and performance are shown and discussed.
This chapter illustrates the scientific environment in which this thesis was written and makes the necessary descriptions of physics.

2.1. Single-particle imaging

This section is in large part based on the sections Introduction and Making the Molecular Movie in the reference 8. All other sources are explicitly cited.

X-ray crystallography has so far been at the forefront of resolving the atomic structures of proteins. To date, the vast majority (~90%) of the structures stored in the Protein Data Base (PDB) have been retrieved by X-ray experiments [9]. However, many interesting proteins are ill-suited for forming crystalline structures and have never been crystallized, so this technique cannot be applied to them. One technique for investigating these structures regardless is applying microscopy techniques (i.e., using lenses) to X-ray imaging. However, the best resolutions achieved by this to date are in the 10-15 nm regime [10, 11], and the design and manufacturing of appropriate lenses have shown to be complicated.

Single particle imaging (SPI) with x-ray beams is another technique for imaging matter at the atomic resolution that applies the concepts of x-ray crystallography to single particles and avoids the limitations of approaches that involve lenses or crystals. Here, an x-ray beam illuminates single particles directly without intermediate lenses, and a position-sensitive detector measures the diffracted x-ray photons. This process produces a continuous diffraction pattern from every single illuminated particle. From these patterns and knowledge of the x-ray field that was incident on the particles,
together with special algorithmic techniques for reconstructing the phase, the object’s structure can—in principle—be uniquely determined and reconstructed.

One problem with imaging biological matter at high resolutions using this technique is the radiation damage induced by prolonged exposure of the sample to high-intensity x-ray radiation\footnote{12}. This damage alters and can destroy the sample. To record measurable diffraction patterns from single particles, however, high intensity is required. A possible solution first brought to attention in 2000 \footnote{7} is to use extremely short x-ray pulses, on the scale of femtoseconds. This has been experimentally shown to be viable at the FLASH facility in Hamburg, Germany \footnote{13}, where such a 25 fs pulse illuminated a sample with known structure. The sample was vaporized entirely in the process, but its structure could nonetheless be reconstructed. The reason this is possible is that the deterioration and destruction induced by radiation damage take some time to manifest, and when the illuminating x-ray pulse terminates within this timeframe, the diffracted patterns are true to the original undisturbed structure. This is the principle of \textit{diffraction before destruction}, which has since been applied in experiments involving imaging with x-ray free-electron lasers (XFELs), which offer exceptionally high intensities at exceptionally short pulse lengths. The first XFEL facilities that enable this kind of imaging research have been built and went into action only within the last decade, like the LCLS (first lasing in 2009), SACLA (first lasing in 2011), and most recently the European XFEL in Hamburg, Germany (first lasing in 2017). They can generate pulses with durations down to 10 fs and up to $10^{13}$ photons per pulse.

However, an atomic-scale resolution has not yet been achieved in SPI. The main bottlenecks in the experimental process that lead to this are the challenges in generating isolated particles and efficiently injecting them into the focus region of the x-ray source \footnote{14}. There are two primary efficiencies to be considered in SPI: the fraction of x-ray pulses that intercept a particle, called the “hit fraction”, and the fraction of injected particles that are intercepted by the x-ray beam, called the “delivery efficiency”. In this thesis, I will try to approach ways to improve these efficiencies via models and simulations.

\section*{2.2. Multiphysics trajectory simulations}

The trajectories single particles take through space determine the focusing of particle beams, and can be analyzed both in qualitative and quantitative
2.2. Multiphysics trajectory simulations

ways. A qualitative look at how particles move through an experimental setup can indicate regions where improvements should be made, taking into account the forces at play. As a simple but contrived example, if unexpected changes of particle paths through a flowing gas are found, one cause might be turbulence of the flow field, which leads to undesirable defocusing of the particle beam. In this case, parameters like the pressure and velocity of the surrounding gas or the geometries of the employed devices should be adjusted.

Analyzing trajectories quantitatively, a useful technique is to let pictures of the particle’s position and velocity distributions at different cross-sections of space emerge. For instance, if the main axis that particles propagate along is called \( z \), one can find the \( z \) position with the highest particle densities by analyzing \( x, y \) slices at different fixed \( z \) positions. One can then—if possible—adjust the position of the devices relative to the region of interest, e.g., the interaction region of an XFEL source, or redesign the devices used. With sufficiently well-designed models of the forces acting on particles, the simulation of trajectories can preface the implementation of experiments and allows both types of analysis to be done in the computer, saving in-lab work hours otherwise spent on incremental improvements and enabling a feedback cycle between model and experiment. New approaches to focusing particle beams based on additional forces can be tested for viability through simulations, before steps towards experimental realization are taken in the lab.

Within this thesis and the developed simulation framework, see Chapter 3, it is assumed that all forces are vectorial, modeled as accelerations, and act independently upon a particle with constant mass, so they can be summed:

\[
\vec{F}_{\text{total}} = m_{\text{Particle}} \cdot \vec{a}_{\text{total}} = m_{\text{Particle}} \left( \vec{a}_1 + \ldots + \vec{a}_n \right)
\]  

(2.1)

Based on this total acceleration field, the ordinary differential equation (ODE) describing Newton’s equation of motion can be solved numerically. Equation 2.2 shows the used definition of this ODE for three dimensions \( x, y, z \), but analogous equations in planar 2D \((x, z)\) or radially symmetrical 2D \((r, z)\) are easily derived. \((x, y, z)\) describes the position and \((v_x, v_y, v_z)\) the velocity. They are considered and calculated together as a six-dimensional phase-space position vector \((x, y, z, v_x, v_y, v_z)^T\). Its derivative is defined as \((v_x, v_y, v_z, a_x, a_y, a_z)^T\), where \(a_x, a_y, a_z\) are the summed acceleration components of the particle at a given time \(t\). Solving this ODE repeatedly with sufficiently small time steps, the trajectories of particles...
are simulated as they evolve through space from initial particle positions and velocities.

\[
y'(t) = f(t, y) \tag{2.2}
\]

\[
y(t) := (x, y, z, v_x, v_y, v_z)^T(t) \tag{2.3}
\]

\[
y(0) := y_0 \tag{2.4}
\]

\[
f(t, y) := (v_x, v_y, v_z, a_x, a_y, a_z)^T(t) \tag{2.5}
\]

## 2.3. Aerodynamical control of particle beams

One approach for focusing beams of nanoparticles is to use forces induced by the flow of a carrier gas through aerosol injector devices with specific geometries and pressures. The gas exerts a drag force on the nanoparticles, leading them to move roughly with the flow of the gas. For SPI experiments, the most widespread devices used for such purposes are aerodynamic lens stacks (ALS) [15]. An ALS is a linear arrangement of tube sections with certain outer radii connected through apertures with smaller inner radii (see Figure 2.1). As the carrier gas flows from one end of the ALS to the other, it is forced to collimate towards the holes of the sections, taking the nanoparticles along with it. Even though the gas expands fully into each subsequent section, the nanoparticles of interest keep on a more focused path due to much higher density and thus inertia compared to the gas particles. There are also other ways of aerodynamic focusing in use, but they will not be discussed here.

### Stokes drag force

The main force leading to the ability of the carrier gas to focus the particle beams is the drag force. The model used, which is applicable for spherical non-interacting particles in laminar flows with Reynolds numbers \( Re \ll 1 \), was developed by Stokes [17] and is defined as follows:

\[
\vec{F}_{\text{Stokes}} = \frac{6\pi \cdot \mu_f \cdot r_p \cdot \Delta \vec{v}_{p,f}}{C_c} \tag{2.6}
\]

where \( \mu_f \) is the dynamic viscosity of the fluid, \( r_p \) the radius of the particle, \( \Delta \vec{v}_{p,f} := v_f(x,y,z,t) - v_p(x,y,z,t) \) is the velocity of the particle relative to the flow velocity at a given position and time, and \( C_c \) is a correction factor.
Figure 2.1.: An idealized illustration of an aerodynamic lens stack with a flow of nanoparticles inside. On the right a laser source is shown interacting with the focused particle beam exiting the lens. *Image courtesy of Nils Roth, reproduced from Figure 2.7 of reference [16]*

derived by Cunningham [18] not present in Stokes’ original publication. Another model based on a modification of the Epstein force that shows promise for being applicable in a wide array of flow regimes and temperatures without requiring such a correction factor has been derived and is also implemented in the program [19].

**Brownian motion**

For nanoparticles, the effect of Brownian motion is non-negligible for most typical temperatures. The model used is taken and slightly adapted from a paper by Li and Ahmadi [20]. The Brownian force per unit mass, i.e., the exerted acceleration, is defined there as a random process based on Gaussian white noise and is dependent on the chosen simulation time step:

\[
\vec{a}_b = \vec{G} \sqrt{\frac{\pi S_0}{\Delta t}} \quad \text{where} \quad S_0 = \frac{216 \cdot \mu \cdot k_B \cdot T}{\pi^2 (2r_p)^5 \rho^2 C_c} \quad (2.7)
\]

\(\vec{G}\) is a vector of three independently identically distributed random numbers from normal distributions with zero mean and unit variance, \(\Delta t\) is the chosen time step, \(\mu\) and \(T\) are the dynamic viscosity and temperature of the gas, \(k_B\) is the Boltzmann constant, \(r_p\) is the particle radius, \(\rho\) is the...
material density of the particle, and $C_c$ is the same slip correction factor as described for Equation 2.6. $S_0$ is the spectral intensity of the random noise. For the special case of radially symmetrical simulations, which are axisymmetrical around the $z = 0$ axis in an $(r, z)$ coordinate system, $\vec{G}$ is a two-dimensional vector. There, the $z$ component is generated the same way as in the 3D case, while the $r$ component is multiplied by a factor of $\sqrt{2}$ [21].

Including Brownian motion like this in a numerical simulation presents two problems. The first is inherent in the process’s randomness, which does not lend itself well to most numerical integration routines: Their convergence criteria depend on the ability to estimate errors based on values of the integrand in a sufficiently small neighborhood, so a force taking on completely random values presents integrators with a seemingly non-convergent problem by breaking this necessary estimation ability. The second problem is that using integrators with adaptive time steps—which is generally well-desirable since it can improve both speed and precision of the integration—leads to a feedback loop: Since $F_b$ is inversely correlated to $\Delta t$, the magnitude of $F_b$ will increase whenever a smaller time step is chosen, leading the integrator to choose a yet smaller time step, increasing $F_b$ again and so on. This presents integrators with a seemingly divergent problem. The way these issues are worked around is that:

1. a macro-timestep $\Delta t$ is set to be constant, and decoupled from any internal adaptive time steps an integrator might choose during the integration of such a macro-timestep, and

2. the Brownian force is, unlike the other forces, not handled by the integrator itself, but added *ad hoc* after every $\Delta t$ integration step, setting $p_{n+1} := p_n + \frac{1}{2} a_b (\Delta t)^2$ and $v_{n+1} := v_n + a_b \Delta t$, where $a_b$ is the acceleration by the Brownian force.

### 2.4. Optical control of particle beams

As has been demonstrated many times within the fields of physics and biology [22][23], the motion of small particles can be controlled using optical forces. Beyond the radiation pressure force [24] which comes to be through impulse exchange of photons and atoms, and the optical gradient force [25], another force exists which shows to be promising for controlling particle beams [14][26]: the so-called photophoretic force [27], which has mostly been researched within aerosol science [28][30].
The photophoretic force manifests itself not directly through photon-atom interactions, but through asymmetrical heating of a nanoparticle surface by incident light. It is thus in fact a thermal force induced by radiation heating. When surrounding gas particles impinge on a particle’s unevenly heated surface, the gas particles hitting regions of higher temperature leave the surface with some of this thermal energy converted into kinetic energy. Through this, they exert a force in the opposite direction of their outwards velocity vector. The nanoparticles are moved by many of these collisions, on average in the opposite direction of the surface-integrated asymmetry vector. Using light beams specifically designed to induce a controlled asymmetrical distribution, it is possible to force a particle beam on more highly focused paths. Diverging beam profiles with a hollow core are suitable, like for Laguerre-Gaussian beams of order $\geq 1$ [31] or (quasi-)Bessel beams [14]. Through the photophoretic force, these beams can control particles and focus them to a central point like a funnel. They are therefore referred to as optical funnels when used for particle focusing.

The theory of the photophoretic force is not yet well-developed. Within this thesis, its numerical calculation and simulation is approached based on a simplified model, presented by Desyatnikov et al [32]. The modeled beam is a monochromatic Laguerre-Gaussian beam of order 1 (LG01 beam). Since this beam’s profile is radially symmetrical, its intensity distribution can be described in $(r, z)$ coordinates:

$$I(r, z) = \frac{P r^2}{\pi w(z)^4} \exp \left( -\frac{r^2}{w(z)^2} \right) \quad (2.8)$$

where $P$ is the total beam power, $w(z) = w_0 \sqrt{1 + z^2 / z_0^2}$ is the ring radius at $z$ for a given beam waist radius $w_0$, and $z_0 = 2\pi w_0^2 / \lambda$ is the diffraction length for the wavelength of the beam $\lambda$.

Through various approximations [32], two expressions for the force’s axial and transverse components can be given. First, a phenomenological coefficient $\kappa [m^{-1}]$ [32], which absorbs thermal and optical parameters of the particle and the surrounding gas is defined. $J_1$ is a scalar parameter measuring a particle-specific asymmetry factor of the force, and is assumed to be $-1/2$ from here on. $\mu_g$ is the dynamic viscosity of the gas, $a$ is the particle radius, $k_{p,g}$ refer to the thermal conductivity of the particle and the gas, respectively, and $T_g$ and $\rho_g$ are the temperature and density of the gas.
Figure 2.2.: The axial and transverse components of the photophoretic force model by Desyatnikov et al [32] as multiples of $\kappa P$. $a$ is the particle radius, $w_0$ is the beam waist radius, $R$ is the magnitude of the radial offset relative to the beam axis, and $z_0$ is the diffraction length of the Laguerre-Gaussian beam.

\[
\kappa := -J_1 \frac{9 \mu_2^2}{2 \sigma T_g (k_p + 2k_g)} \quad (2.9)
\]

The transverse force component $F_R$ is then approximated as:

\[
F_Z(z) = \kappa P f_Z \left( \frac{a^2}{w(z)^2} \right) \quad (2.10)
\]
\[
f_Z(t) = 1 - (1 + t) \exp(-t) \quad (2.11)
\]

The approximation for the axial force component $F_Z$ is as follows:

\[
F_R(r, z) = -\frac{2\kappa P}{\pi} \int_0^a \int_{\sqrt{a^2-x^2}}^{\sqrt{a^2-x^2}} f_R(x, y, z, r, a) \, dy \, dx \quad (2.12)
\]
\[
f_R(x, y, z, r, a) = \frac{y x^2 + (y + r)^2}{w(z)^4} \frac{\exp \left( -\frac{x^2 + (y + r)^2}{w(z)^2} \right)}{\sqrt{a^2 - x^2 - y^2}} \quad (2.13)
\]

Implementing this force, curves for varying particle radii $a$ and radial offsets $R$ are drawn in Figure 2.2, which match the ones published by Desyatnikov et al [32].
CMInject: An extensible framework for trajectory simulations

To simulate particle trajectories with a specialization on the focusing of nanoparticle beams, the Python simulation framework Controlled Molecule Injection (CMInject) was developed, based on an existing program developed within the Controlled Molecule Imaging (CMI) research group [33]. The central goal of the development of a new version is to provide users with an extensible object-oriented framework for defining and running such simulations, rather than to provide a program for a single kind or few kinds of experiments. The software can simulate trajectories within arbitrary spatial dimensions, as long as all objects comprising a virtual experiment can work with the given dimensionality. It is designed to run arbitrary multiphysics simulations, e.g., simple fluid dynamics simulations, photon trapping force simulations or even nanoparticle sorting experiments like done with a Stark deflector [34]. The former two kinds of experiments were implemented and simulated for this thesis. The development, improvement, and result and performance analyses of this framework constitute the major part of the work done for this thesis.
3. CMInject: An extensible framework for trajectory simulations

3.1. Overview

CMInject is a computational framework for defining and running multi-physics Monte Carlo particle trajectory simulations, for storing the results of such simulations, and for visualizing and analyzing these results. It was developed in the Python programming language, using libraries for scientific computing like scipy, numpy and matplotlib. It is compatible with Python versions 3.6 and upwards. The core features that CMInject offers to users and developers are as follows, each corresponding to a subsection of this section:

1. A set of abstract base classes corresponding to real-world experimental objects like particles, devices containing acceleration fields and detectors, for developers to derive new subclasses from and define own simulation experiments with

2. An abstract base class for virtual experiment setups which combine experimental objects into a virtual experimental setup that can be simulated with varying parameters

3. The main program, which numerically simulates the propagation of particles within a given experiment setup

4. Capabilities for storing, visualizing and analyzing simulation results

An abstract base class (ABC), in this context, refers to a concept implemented in Python. ABCs “define a minimal set of methods that establish the characteristic behavior of the type” [35], meaning they, without necessarily providing a default implementation, define a class interface that concrete subclasses must adhere to. Subclasses of any ABC that do not fully implement this defined API can not be instantiated as objects. The way this concept is used within CMInject is to provide developers with an enforced blueprint for definitions of experimental objects and setups. Doing so allows the core of the program to work with the assumptions made by those abstract base class definitions, and in effect, enables the provided extensibility.

For all abstract base classes and setups, sensible, concrete definitions were implemented and are provided with the software. Further concrete definitions can—based on the abstract definitions—be straightforwardly implemented and integrated into a simulation. Offering such default implementations while allowing new ones to be added based on blueprint classes is intended to appeal both to users who want to run simulations and to developers who want to implement new models.
3.2. Object class definitions

Herein, the most important class definitions in CMInject are described. Information on all other definitions can be retrieved from CMInject’s documentation. It is a convention in this section that uppercase nouns refer to a class defined in the framework, and lowercase nouns refer to the corresponding physical objects. The interfaces defined through these class definitions are listed in Appendix A.

NDimensional

All classes mentioned below inherit from the NDimensional class. Instances of NDimensional’s subclasses must provide a method to set the dimension of space on the object. They should either take action, e.g., storing this number of dimensions or exchanging a method implementation, or raise an error if the object is not compatible with the given number of spatial dimensions.

Particle

Particles are the objects whose trajectories are simulated. Within CMInject, they are considered to be just data containers that fully describe the particle through a collection of properties (like their position, velocity, radius, and others), and beyond this, they should only offer convenience methods for read/write access to this data. To provide an encapsulated program structure, they ought to be unaware of any objects that affect them, meaning they do not store or use references to such objects. The entire particle description and particle history that will be stored for one specific particle in a simulation’s output file must be stored on the corresponding Particle instance.

Field

Fields, in CMInject, refer to acceleration fields that interact with particles. A Field subclass must provide a method to calculate the acceleration for a given particle instance at a given time. Fields are, with one minor exception (PropertyUpdaters), the only objects that affect a particle’s current position and thus determine its trajectory.
3. CMInject: An extensible framework for trajectory simulations

Boundary

Boundaries describe spatial boundaries. A Boundary subclass must provide a method to check if a Particle is within the boundary or not, which can implicitly describe arbitrary boundary geometries, including infinite extent in at least one dimension. Using Boundary instances, the trajectory simulation of a particle is stopped when it leaves the experimental region of interest.

Device

Devices combine one or multiple Fields and one Boundary into objects akin to real-world laboratory devices like a steel tube containing a flowing gas, a laser, or a full aerodynamic lens stack.

Detector

Detectors are simply objects that can tell whether and where a Particle has hit them, and a Detector subclass must provide two methods for exactly those purposes. While defining anything other than a detector at a given Z position was not deemed necessary during this thesis, this broad definition also allows for arbitrary detector geometries.

Source

Sources describe initial particle distributions at the beginning of a simulation. They must provide a method to generate such a distribution in the shape of a list of Particle instances.

PropertyUpdater

PropertyUpdaters are very generic objects that will be called after each simulation time step and can update arbitrary properties of the particles. Many different tasks can be implemented with PropertyUpdaters, like storing a particle’s trajectory, calculating the effect of Brownian motion on position and velocity (see section 2.3), or the time-dependent exponential cooling of a Particle.

Experiment

The Experiment class is instantiated with one list each of Devices, Sources, Detectors, and PropertyUpdaters, along with other parameters relevant for
3.3. Defining and running virtual experiment setups

Setup subclasses define virtual experiment setups. They must provide one method that returns an argument parser, which should define and read in all parameters of the setup that can be changed, like initial position/velocity distributions, the pressures of different sections of the experiment. They must also provide a method that takes these parsed parameters and returns an Experiment instance.

To simulate an experiment setup, the user must pass the fully qualified name of the setup class to the main program, cminject, as a command-line argument. When cminject runs with a given setup class, it parses the arguments that are common to all simulations (e.g., the number of particles), and forwards all other arguments to the setup class’s parser. This design was implemented to require only one main program to run different setups with different parameters. To make the available parameters and their usage clear to the user, the help text displayed by cminject includes the help strings of both the main program’s arguments and the setup-specific arguments. This help is displayed when either the -h flag argument is passed, or when unknown or wrong arguments are given by the user. Code for an example setup used in section 4.4 is listed in Appendix B.

3.4. Implementation of particle propagation

Numerical integration

Particles are propagated by the software by numerical integration of the ODE describing Newton’s equation of motion. CMInject approximates solutions to this ODE using a numerical ODE solver routine called LSODA, to which the Python package scipy.integrate provides an interface. This solver does automatic method switching between the nonstiff
3. CMInject: An extensible framework for trajectory simulations

Adams method and the stiff Backwards Differentiation Formulas (BDF) method, with relatively low computational overhead involved for switching, as first detailed in [36]. While there might be better or faster choices if the numerical conditions are well-known for all points in a specific simulation setup, this low-overhead method switching makes LSODA a good default choice for a framework where arbitrary forces and conditions can be defined.

Parallelization

Since the single particles we are interested in are assumed to be non-interacting within the injection pipelines, the computational task at hand is “embarrassingly parallel”, which is a term used in computer science to refer to problems where each process can perform its own computations without the need to communicate with any other process. The computation can thus be parallelized in a simple fashion. This is done using a process pool, multiprocessing.Pool, from the multiprocessing module which is included with the Python core. The way this works is that the main process starts a number of \( N \) subprocesses, where \( N \) should at maximum be the number of available CPU cores: Having one CPU core handle the simulation of two or more particles at once will most likely not enhance performance but be detrimental to it due to necessary interrupts, context switching and cache misses. The main process constructs all initial object instances and passes a serialized copy of all of them to each subprocess. It then manages a queue of tasks, each task corresponding to one Particle instance which is in its initial state, i.e., having all initial property values as assigned by the Source that generated the Particle instance. The tasks are taken from the queue by the subprocesses each time a subprocess becomes available for work, which is once at initialization and each time it completes its current task. The fully simulated Particle instance resulting from the previously completed task is communicated back to the main process. After all tasks are completed, the main process stores the resulting Particle instances in an output file that is described in section 3.5.

3.5. Output storage, analysis and visualization

CMInject offers, at the time of writing, a structured HDF5 [37] file as its only output format. Tools for processing and analyzing this result data are provided as separate programs in CMInject. A program is provided
3.5. Output storage, analysis and visualization

to visualize the output in different ways; currently a trajectory visualization and a detector distribution visualization are offered. Output files are written by subclasses of ResultStorage, and the visualizations are implemented as subclasses of Visualizer. Both of these classes are abstract base classes with a well-defined interface. Other ways to store, analyze, and visualize results can thus be easily implemented and added to the CMInject framework.

The HDF5 result format

HDF5 result files are generated by HDF5ResultStorage based on lists of Particle instances that are in a fully simulated state, meaning they’ve stored their initial, final and detected positions and properties. HDF5 allows storing numerical data in a tree-like format with named nodes. This is used to write the output as a tree of named entries, illustrated in Figure 3.1. The detector hits key has \( m \) subentries, where \( d_0, \ldots, d_m \) are identifiers of the detectors in the simulation that have registered at least one hit. The particles key has \( n \) subentries, where \( n \) is the number of simulated particles. Each particle key has subentries for the initial position, the final position, and the trajectory of the particle (if the program was instructed to store trajectories). The trajectory entries and the sub-entries of detector hits are 2-D arrays, and they are annotated with metadata that names each column in these arrays for easier analysis and interpretation of the results. HDF5ResultStorage offers methods to retrieve this data from a previously written result file in read-only mode.

Result processing and analysis

Two programs are provided for further processing and analysis of result data. cmInject_reconstruct-detectors reconstructs measured quantities at virtual detectors with a fixed \( z \) position based on (linear or cubic spline) interpolation along the stored trajectories. This can be useful for not having to run an entire simulation again only to analyze the particle property distributions at additional positions in space. Furthermore, animated movies of particle property distributions as they evolve through space can be created by reconstructing a large number of detectors.

cmInject_analyze-asymmetry analyzes and prints information about the asymmetry of a two-dimensional distribution, like \( x, y \), at each stored detector. The output can be configured to be either an easily human-readable format, or machine-readable CSV. The process to calculate this information
3. **CMInject: An extensible framework for trajectory simulations**

output_file.h5
  ├── detector_hits
  │       ├── d₀
  │       │   ├── ...
  │       │   └── dₘ
  │       └── particles
  │           └── 0
  │               ├── initial_position
  │               │   └── final_position
  │               └── trajectory
  │                   └── ...
  └── n

Figure 3.1.: The shape of the data tree that is stored in result HDF5 files. There are \( m \) different detectors and \( n \) different particles in this example.

is provided in more detail along with an analysis in [subsection 4.1.2](#).

**Result visualization**

The program `cminject_visualize` offers several options for using two different kinds of visualization code to get an overview of simulation result files, using the two Visualizer subclasses `TrajectoryVisualizer` and `DetectorHistogramVisualizer`. `TrajectoryVisualizer` shows the particles’ initial, detected, and final positions as colored scatter plots, and their trajectories as curves. It was used for instance to create the plots in Figure 4.1. By default, each trajectory is plotted with a single color; optionally, each trajectory can be plotted as multiple colored line segments, the color of each segment indicating the particle’s local velocity.

`DetectorHistogramVisualizer` plots (1D or 2D) distribution histograms for all stored detectors. Each row corresponds to a detector, and each column corresponds to a particle property pair chosen by the user. The visualizer decides automatically whether to plot a 1D or 2D histogram for each pair based on whether all values in one dimension are equal or not.
This chapter shows qualitative and quantitative analyses made on results generated by CMInject and compares them against results from existing similar programs. Almost all simulations described are based at least in part on one or multiple flow field files, which are multidimensional grids of the pressure and flow velocities in each considered direction, at equidistant sample points. These files were originally generated by a finite-element solver in the COMSOL MULTIPHYSICS software and exported as a simple text output format. This text format was subsequently converted to a file in the HDF5 format with the tool cminject_txt-to-hdf5, one of the runnable scripts coming with CMInject, to remove the overhead of reading in large text files representing grids of numerical data.

4.1. Result analysis of toy problem simulations

To verify the plausibility of the results generated by CMInject both for axially symmetrical (2D/3D) and asymmetrical (3D) conditions, the program was first run for a radially symmetrical problem, comparing a 2D and 3D simulation of the same problem. Subsequently, it was verified that asymmetrical 3D problems can be simulated appropriately as well. This section shows the obtained results.

4.1.1. Radially symmetrical flow simulations in 3D and 2D

To show that CMInject outputs matching results for axially symmetrical problems in 2D and 3D without any additional effort required from users, a
2D version and a 3D version of the same axially symmetrical flow field were generated and results compared. For the 2D case, the field and a trajectory simulation through the field are shown in Figure 4.1. Regions with \( p = 0 \) imply the field’s boundary, so the simulation of a particle stops when it enters such a region. The corresponding 3D field is created by revolving this 2D field around the \( z \) axis. Histograms for detected \( r \) positions at varying \( z \) distances from the start position (\( z = 0 \)) are shown in Figure 4.2. It is clear that when ignoring Brownian motion, the 2D and 3D simulation results fully agree. When simulating Brownian motion, the 2D simulation seems to leave particles moving closer to \( r = 0 \), smearing the initially chi distributed \( r \) towards 0, and giving the appearance of a more focused beam than it is the case in 3D. The models used for both simulations were taken from literature with good theoretical results [20, 21], see section 2.3, so reasons for this should be further investigated.

4.1.2. Asymmetrical 3D flow simulations in low temperatures

Experiment conditions are not necessarily radially symmetrical. Thus, the software should be able to handle fully asymmetrical problems just as well. Through comparing simulations of differently asymmetrical 3D fields and considering the asymmetry of the final particle distribution when the initial distribution is symmetrical, it can be ensured that this is the case.

Herein, the considered problem is not as simple as the one described for 2D. It is a partial simulation of an experiment where particles are injected into a cryogenically cooled buffer gas cell (BGC) with a gas temperature of \( T = 4.0 \) K as described by Singh et al [38]. Since cold helium gas constantly flows into the buffer gas cell from only one side, the gas flow and thus the particle paths are asymmetrical. Four simulations were run, taking the same simulated flow field geometry for four different Helium gas inlet flow rates in standard cubic centimeters per minute (sccm). Gaussian distributions were assumed for the initial particle positions and velocities as listed in Table 4.1. \( 1 \times 10^6 \) Polystyrene particles were simulated for each flow rate value, and were assumed to have a fixed radius \( r_p = 500 \) nm and consist of Polystyrene, so that \( \rho_p = 1050 \) kg m\(^{-3}\) and thus \( m_p \approx 550 \) fg for all particles.

The asymmetry of the \( x \) versus the \( y \) position distributions of particles at a given \( Z \) position was determined by taking all detected points at a virtual detector positioned 0.6 mm away from the BGC outlet and calculating a discrete approximation to the inertia tensor \( \Theta \) as defined in Equation 4.1, determining its eigenvalues \( \lambda_1 \) and \( \lambda_2 \), and computing a measure of asym-
4.1. Result analysis of toy problem simulations

Figure 4.1: (a) The radially symmetric toy problem field as an intensity plot of the pressure $p$ and a vector field plot of the flow speeds $v_r$ and $v_z$. (b, c, d) Simulation results plotted over the field as trajectory curves of 100 randomly sampled particles, with velocities encoded by a color map. The carrier gas is $N_2$ at $T = 293.15$ K. Virtual detectors were inserted at $z \in \{0 \text{ cm}, 1 \text{ cm}, 1.05 \text{ cm}, 1.1 \text{ cm}, 2 \text{ cm}\}$, and the detected particle positions are marked by magenta dots. (b) shows the trajectories for $r_p = 100 \text{nm}$ particles simulating Brownian motion, (c) for $r_p = 5 \text{nm}$ particles ignoring Brownian motion, and (d) for $r_p = 5 \text{nm}$ particles simulating Brownian motion.

<table>
<thead>
<tr>
<th>$x$ [m]</th>
<th>$y$ [m]</th>
<th>$z$ [m]</th>
<th>$v_x$ [m/s]</th>
<th>$v_y$ [m/s]</th>
<th>$v_z$ [m/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>0</td>
<td>0</td>
<td>4.8×10^{-3}</td>
<td>0</td>
<td>-43.0</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>2×10^{-4}</td>
<td>2×10^{-4}</td>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 4.1: The mean ($\mu$) and standard deviation ($\sigma$) of the assumed initial Gaussian distributions of the particle positions and velocities for the 3D buffer gas cell problem simulation.
4. Simulations and results

(a) Without Brownian motion

(b) With Brownian motion

Figure 4.2.: Comparing the detected $r$ positions at different $z$ positions throughout a radially symmetrical toy problem simulation for $10^4$ particles in both 2D and 3D, once (a) with Brownian motion ignored and once (b) with Brownian motion simulated. Lines are plotted for easier visual comparison and pass through each bin’s $x$ midpoint and its counted occurrences. The 2D case is barely visible in (a) because of the good match of the 2D and 3D distributions there.

Theoratically, the final asymmetries $\alpha_f$ determined at the virtual detector should be compared against the initial asymmetries $\alpha_s$; however, since $\mu_x = \mu_y$, $\sigma_x = \sigma_y$, $\alpha_s$ is 0 and might deviate numerically from this value only due to the initial distribution being a finite sample. All $\alpha_s$ are thus dis-
regarded, and the different $\alpha_f$ are plotted in Figure 4.3 for the different flow rates. It is visible that increasing the flow rate increases the asymmetry by a large factor, being almost 0 for 10 sccm and around 0.4 for 70 sccm. It can be concluded that asymmetric 3D problems can be, at least qualitatively, simulated correctly.

Comparing against the asymmetries calculated from experimental data, the simulation data shows to be less trustworthy. The issue with making a proper comparison is that initial conditions are not known, specifically the position and velocity distributions of the particles entering the BGC. We are thus comparing purely induced asymmetry based on symmetrical initial conditions (as in the simulation) with measured final symmetry, which might well be influenced by asymmetry already present in the particle beam before the BGC. Iteratively trying different asymmetrical initial distributions to find the best match between experiment and simulation should be worthwhile, leading either to a conclusion about the simulation’s applicability or a better idea of the initial particle distribution, but this is not investigated here due to time constraints.
4. Simulations and results

4.2. Comparisons against existing programs

4.2.1. The previous version of CMInject

Since the developed software framework was based on an existing program which had previously been verified against at least one experiment, it was necessary to compare the results returned by the old and new versions. This way, correctness of the results can be benchmarked against software that has previously been used and shown to return plausible results.

As the existing program was written and used for 3D simulations, the same 3D buffer gas cell flow field as described in subsection 4.1.2 was used, taking the flow field with a flow rate of 50 sccm to compare the results of a simulation of $10^4$ particles starting from the exact same positions and velocities in both programs. Brownian motion was again disregarded, both due to the nature of the problem and to this model missing completely from the old program. The final particle distributions for both simulations and the normed differences of the final positions are shown in Figure 4.4. It is clear that the shown distributions match up to numerical error, with the exception of a few outlier particles (14 in total). These were found to have stopped propagating earlier in the new version, which is due to slightly different conditions for continuing integration at flow field boundaries in both versions.

4.2.2. particleTracing.py: A team-internal tool

particleTracing.py is a tool built internally at the CMI group by Nils Roth [19]. It was designed and used specifically for simulations of aerodynamic lens stacks in radially symmetrical problems, and propagates particle in $(r, z)$ coordinates. Since these lens stacks are typically symmetrical up to a certain degree of experimental inaccuracy, this is a good approximation. The program has been in use within the team for some time and has calculated a few simulational results that fit experiments well; it was for instance used within the team to incrementally optimize an ALS consisting of 5 segments. This ALS is the one simulated in section 4.4. Nonetheless, a goal is to replace it with CMInject, as it is lacking in extensibility and readability with all code being contained in a single script, and as it can not simulate forces other than a drag force and the Brownian force. Before being able to replace this program with CMInject, it is useful to compare their results (and performance, see subsection 5.3.2). The final particle distributions from both programs are shown in Figure 4.5. It can be seen that
4.2. Comparisons against existing programs

Figure 4.4.: (a) Final particle distributions of the existing version of CMInject (left) and the newly developed version of CMInject (right). Distributions are shown as 2D histograms using a logarithmic scale for the particle counts. Both histograms share their bin edges. (b) The distribution of distances $d := |(x_n, y_n) - (x_o, y_o)|$, where $(x_o, y_o)$ and $(x_n, y_n)$ are the coordinates from the old and new program, respectively.

results are generally in agreement, but the final distribution retrieved from CMInject seems to be slightly more spread out. A likely reason for these differences is the method of interpolation implemented in both programs. To interpolate on a regular grid, which is how a continuous drag force field is approximated, CMInject uses multilinear interpolation—bilinear interpolation in this 2D case—while particleTracing.py uses cubic spline interpolation. As the interpolation error from cubic splines is generally much lower than that from linear interpolation, this interpolation method should be added to CMInject to improve result accuracy.
4. Simulations and results

![Particle distribution plots](image)

Figure 4.5.: Final particle distributions for an ALS simulation by particleTracing.py as 2D histograms of $x,y$ (top) and a 1D histogram of $r$ (bottom). For the plots at the top, each $r_i$ in the radial $r$ distribution was taken and rotated about a uniformly random angle $\phi_i \in [0, 2\pi]$ to retrieve a $x/y$ distribution assuming radial symmetry. At the bottom, the $r$ distributions are shown as probability density histograms with a kernel density estimation. Coordinates are in multiples of $1\,\mu\text{m}$.

4.3. Photophoretic focusing

As described in section 2.4, the photophoretic force can be used to focus particle beams. This section will show trajectory plots and particle distribution plots resulting from simulations of this force. The particle properties having an effect within these simulations are the particle radius $r_p$ and density $\rho_p$, as well as the particle thermal conductivity $k_p$. As optical properties like the refractive index influence the strength force in complex ways that are yet hard to estimate especially for particles close to the light wavelength, changes in force due to this are disregarded within this thesis. Since it influences the magnitude of the force, the difference in absorption efficiency from simple approximations and from an exact solution through
Mie theory is discussed in Appendix D

A monochromatic LG01 laser beam with wavelength $\lambda = 532\text{nm}$ positioned at the origin of an $(r, z)$ coordinate system with a beam waist radius of $w_0 = 8\mu m$ and a fixed beam power $P$ was assumed. $10^5$ polystyrene particles with $r_p = 220\text{nm}$, $\rho_p = 1050\text{kg/m}^3$ and $k_p = 0.030\text{W/m·K}$ were simulated entering this beam at a fixed $z = 5\text{mm}$ away from the origin with the $r$ position projected from two i.i.d. random Gaussian distributions with $\mu_{x,y} = 0, \sigma_{x,y} = 20\mu m$. The particles were assumed to be moving through a uniform flow field of nitrogen at $T = 293.15\text{K}$ and $p = 100\text{mbar}$ with $v_r = 0, v_z = -10\text{m/s}$ everywhere and Brownian motion enabled.

A trajectory plot qualitatively illustrating the particle paths and detectors is shown in Figure 4.6 for laser powers $P_1 = 0.05\text{W}$ and $P_2 = 1\text{W}$. The corresponding histogram plots for the detectors at the $z$ positions $z_d \in \{3\text{mm},1.5\text{mm},0.0\text{mm},-1.0\text{mm}\}$ away from the beam center are shown in Figure 4.7. It can be seen that with the chosen beam waist radius $w_0$ and initial starting distance $z = 5\text{mm}$, particles far away from $r = 0$ are not captured by the laser at all, and particles at an intermediate distance from 0 are actually deflected away. This observation is also qualitatively illustrated in Figure 4.8 for a simpler simulation. Nonetheless, the particle density close to $r = 0$ is increased significantly, and the fraction of captured particles grows as the laser power is increased. It is also visible that while the particle density around $r = 0$ is highest at $z = 0$, this density is maintained well at $z = -1.0\text{mm}$, and this holds especially well for $P = 1\text{W}$.

It should be noted that light absorption—and as such, the real effect of the photophoretic force—is not approximated well here since Polystyrene is highly transparent at $\lambda = 532\text{nm}$, but the model assumes a fully absorbing sphere. This was not done because of the theoretical and numerical difficulty of determining the true absorption efficiency, which in reality varies not only with particle size and material, but also with the beam profile and polarization due to the varying absorption at different light polarization angles and incident angles, see also section 2.4 and Appendix D.

4.4. Multiphysics experiment setups

One of the goals of this thesis was to provide a framework for simulations in which physical forces can act independently, but can also interact with each other. An exemplary experimental setup combining two separate pre-calculated flow fields, a geometric section in which no forces are active, and a photophoretic laser force field from a LG01 vortex beam is shown
4. Simulations and results

Figure 4.6.: Trajectory plots of a particle simulation that combines a photophoretic LG01 vortex laser beam with a uniform fluid flow field moving at $-10 \text{ m/s}$, see section 4.3. The decadic logarithm of the local laser intensities at $P = 0.05 \text{ W}$ is shown in the background; at 1 W, these intensities would scale by $\log_{10}(1/0.05) \approx 1.3$. The particle paths and detected positions in black are for $P_1 = 0.05 \text{ W}$, the ones in white for $P_2 = 1 \text{ W}$. (a) shows these trajectories for $r \in [0 \text{ mm}, 0.1 \text{ mm}]$, (b) shows the same plot zoomed in on $r \in [0 \text{ mm}, 0.01 \text{ mm}]$. Histograms of detected $r$ positions are shown in Figure 4.7.

Figure 4.7.: Histograms of the particle $r$ distributions for a photophoretic focusing experiment, see Figure 4.6, at four detectors with varying distances $d$ from $z = 0$. The evolving increase in particle density close to the origin is visible. (a) shows the histograms for $P_1 = 0.05 \text{ W}$, (b) for $P_2 = 1 \text{ W}$. The distribution at $d = -1.0 \text{ mm}$ is obscured close to 0 in (b) because it matches the distribution at $d = 0 \text{ mm}$ closely there.
**4.4. Multiphysics experiment setups**

Figure 4.8.: A qualitative trajectory plot for a photophoretically active LG01 vortex laser beam illustrating both focusing and deflection depending on the offset from the beam axis at \( x = 0 \).

Together with simulation results in Figure 4.9, particles were assumed to be \( r = 50 \text{ nm} \) gold nanospheres entering the first skimmer with an initial \( r \) distribution derived from two independent equally distributed Gaussian distributions for \( x \) and \( y \) with \( \mu_{x,y} = 0, \sigma_{x,y} = 3 \text{ mm} \). The code defining this setup is listed in the appendix, see Listing 1.

To model the interaction of the photophoretic force with the ALS flow field close to the ALS exit, the interpolated pressure component of the flow field is used by the laser field to calculate first the local gas density based on the ideal gas law \( p_g M_g = \rho_g R T_g \), and then calculate a local \( \kappa \) (see Equation 2.9). In the region where the flow field is no longer defined, a constant outer pressure of \( 5.9 \times 10^{-4} \text{ mbar} \) is assumed to calculate \( \rho_g \).

Since the purpose of this section is only to demonstrate the possibility of defining such setups, the physical plausibility of the simulation results is disputable. \( \kappa \) scales inversely with the gas density in Desyatnikov’s photophoresis model. As this would lead to an infinite force at zero density, it can be assumed that the model is only applicable up to a certain minimum pressure. The high thermal conductivity of gold would, in reality, likely also be detrimental to the magnitude of the photophoretic force, as the surface thermal distribution asymmetry required for photophoresis to take place might be equilibrated too quickly for gold nanoparticles.
4. Simulations and results

(a) 

(b) 

(c) 

Figure 4.9.: A multiphysics setup combining a fluid flow drag force and photophoresis, see section 4.4, and simulation results. Gold nanoparticles with $r_p = 50\,\text{nm}$ were simulated moving through the geometry shown in (a) from left to right, through two skimmers (red/grey) and an ALS (blue) as well as a 532 nm LG01 laser beam device (green) attached after the ALS exit. (b) shows qualitatively the trajectory differences when the laser device after the ALS exit is turned off (left) and on (right). Virtual detectors were inserted at the positions marked by dashed lines. In (c), the $r$ distribution at the inserted detectors is shown for the laser attached after the ALS exit turned off ($\equiv 0\,\text{W}$) and turned on at 1 W power. Particles further away than 5 $\mu$m from the origin are not considered in these distributions.
CMInject’s time performance for example simulations is shown in section 5.1. Its performance is compared both to the previously developed version as well as another tool that was developed within the CMI group to run trajectory simulations for a specific problem kind. All time and memory measurements were taken with the GNU time utility. A further suggestion to possibly dramatically improve the performance of the software is discussed in chapter 6.

5.1. Runtime performance overview

Due to the high variety of numerical conditions in different problems, experiment setup sizes, and simulated forces, a fully general statement about CMInject’s performance cannot be made. Nonetheless, to give an overview of performance for practical reasons, simulation runtimes are listed for different simulations, some of them executed on nodes of the Maxwell computing cluster at DESY, and some executed on a “MacBook Pro (Retina, 15-inch, Late 2013)” computer, to show the kinds of problems one can simulate both with a cluster and a desktop computer. Similar simulations for larger numbers of particles can be extrapolated from this data since the program has expected $O(n)$ runtime behavior, i.e., its runtime scales linearly with the number of particles, see subsection 5.3.1 and subsection 5.3.2. Table 5.1 lists performance measurements on Maxwell for the problem described in subsection 4.1.2 varying the flow rate. To also give an idea of performance on a desktop machine, Table 5.2 lists measurements on a “MacBook Pro (Retina, 15-inch, Late 2013)” computer for the radially sym-
5. Performance evaluation

<table>
<thead>
<tr>
<th>Flow rate</th>
<th>Clock time</th>
<th>User time</th>
<th>Kernel time</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 10^6, No trajectories saved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 sccm</td>
<td>01:34 h</td>
<td>73:33 h</td>
<td>00:25 h</td>
<td>4.78 GiB</td>
</tr>
<tr>
<td>30 sccm</td>
<td>01:56 h</td>
<td>103:03 h</td>
<td>00:36 h</td>
<td>4.78 GiB</td>
</tr>
<tr>
<td>50 sccm</td>
<td>02:06 h</td>
<td>102:38 h</td>
<td>02:06 h</td>
<td>4.78 GiB</td>
</tr>
<tr>
<td>70 sccm</td>
<td>02:20 h</td>
<td>118:54 h</td>
<td>01:54 h</td>
<td>4.76 GiB</td>
</tr>
<tr>
<td>n = 10^5, Trajectories saved</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 sccm</td>
<td>00:14 h</td>
<td>07:33 h</td>
<td>00:09 h</td>
<td>40.50 GiB</td>
</tr>
<tr>
<td>30 sccm</td>
<td>00:16 h</td>
<td>09:54 h</td>
<td>00:10 h</td>
<td>44.75 GiB</td>
</tr>
<tr>
<td>50 sccm</td>
<td>00:17 h</td>
<td>10:47 h</td>
<td>00:10 h</td>
<td>49.88 GiB</td>
</tr>
<tr>
<td>70 sccm</td>
<td>00:19 h</td>
<td>11:55 h</td>
<td>00:13 h</td>
<td>56.09 GiB</td>
</tr>
</tbody>
</table>

Table 5.1.: Time and memory measurements for simulations of an asymmetric 3D buffer gas cell, see subsection 4.1.2. All simulations were run on nodes of DESY’s Maxwell cluster with 64 virtual cores using Intel Xeon E5-2689 processors. “Max. memory” refers to the maximum resident set size.

metrical toy problem described in subsection 4.1.1, comparing 2D against 3D and Brownian motion being enabled and disabled. Table 5.3 shows measurements for photophoretic focusing simulations, see section 4.3, run on a DESY Maxwell node, varying the length of the simulated z section of space. In Table 5.4, performance of the multiphysics simulation described in section 4.4 is evaluated, changing the number of particles and whether the laser was turned on or off.

5.2. Multiprocessing overhead

When parallelizing a program, there is a performance overhead due to the additional need to coordinate between several threads or processes, and the need to communicate data between processes. CMInject is parallelized using multiple worker processes, each retrieving, then simulating and returning to the main process one particle at a time. Communication between any two worker processes does not occur. The way these particles are communicated between the workers and the main process is through “(un)pickling”, a Python term for (de)serialization of Python objects, which are communicated through special in-memory files at the beginning and at
### 5.2. Multiprocessing overhead

<table>
<thead>
<tr>
<th>( n )</th>
<th>2D/3D</th>
<th>Clock time</th>
<th>User time</th>
<th>Kernel time</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^4 ) 2D</td>
<td>00:04:49 h</td>
<td>00:35:47 h</td>
<td>00:00:10 h</td>
<td>0.83 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^4 ) 3D</td>
<td>00:07:05 h</td>
<td>00:50:57 h</td>
<td>00:00:17 h</td>
<td>0.92 GiB</td>
<td></td>
</tr>
</tbody>
</table>

**Brownian motion enabled**

<table>
<thead>
<tr>
<th>( n )</th>
<th>2D</th>
<th>Clock time</th>
<th>User time</th>
<th>Kernel time</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^4 ) 2D</td>
<td>00:07:53 h</td>
<td>00:54:13 h</td>
<td>00:00:21 h</td>
<td>0.71 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^4 ) 3D</td>
<td>00:08:07 h</td>
<td>01:00:28 h</td>
<td>00:00:17 h</td>
<td>1.01 GiB</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2.: Time and memory measurements for simulations of a 2D radially symmetrical toy problem, see subsection 4.1.1. All simulations were run for \( 10^4 \) particles on a “MacBook Pro (Retina, 15-inch, Late 2013)” computer, using 8 virtual CPU cores with HyperThreading on 4 physical cores. “Max. memory” refers to the maximum resident set size.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \ell )</th>
<th>Clock time</th>
<th>User time</th>
<th>Kernel time</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^3 ) 3 mm</td>
<td>00:24 h</td>
<td>25:05 h</td>
<td>00:01 h</td>
<td>0.16 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^3 ) 6 mm</td>
<td>00:47 h</td>
<td>49:31 h</td>
<td>00:01 h</td>
<td>0.17 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^4 ) 6 mm</td>
<td>08:13 h</td>
<td>519:29 h</td>
<td>00:23 h</td>
<td>0.40 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^3 ) 11 mm</td>
<td>01:31 h</td>
<td>95:10 h</td>
<td>00:02 h</td>
<td>0.18 GiB</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3.: Time and memory measurements for a photophoretic focusing experiment described in section 4.3. All simulations were run on nodes of DESY’s Maxwell cluster with 64 virtual cores using Intel Xeon E5-2689 processors. \( \ell \) is the simulated problem length, and “Max. memory” refers to the maximum resident set size.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Laser</th>
<th>Clock time</th>
<th>User time</th>
<th>Kernel time</th>
<th>Max. memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^3 ) Off</td>
<td>00:45 h</td>
<td>45:04 h</td>
<td>00:11 h</td>
<td>6.51 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^3 ) On</td>
<td>02:54 h</td>
<td>91:00 h</td>
<td>00:05 h</td>
<td>– Data missing –</td>
<td></td>
</tr>
<tr>
<td>( 10^4 ) Off</td>
<td>12:59 h</td>
<td>410:52 h</td>
<td>00:19 h</td>
<td>62.73 GiB</td>
<td></td>
</tr>
<tr>
<td>( 10^4 ) On</td>
<td>16:23 h</td>
<td>1028:45 h</td>
<td>01:25 h</td>
<td>62.83 GiB</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4.: Time and memory measurements for the multiphysics experiment setup described in section 4.4. All simulations were run on nodes of DESY’s Maxwell cluster with 64 virtual cores using Intel Xeon E5-2689 processors. Memory usage was not measured.
5. Performance evaluation

While for a large number of CPU cores, the performance gained by splitting the problem into pieces should dominate the overhead induced by parallelized organization, it is nonetheless relevant to get a picture of the parallelization overhead as it can potentially be optimized. To get a rough picture of the overhead, simulations with equivalent distributions were run for varying numbers of CPU cores, first varying \( n_{\text{CPU}} \in \{1, 2, 4, 8, 16, 32, 64\} \) processes for a fixed number of \( n_{\text{Particles}} = 1000 \), and then using \( n_{\text{CPU}} \in \{1, \ldots, 64\} \) for \( n_{\text{Particles}} = 100 \). The results are shown in Figure 5.1 as the total simulation time multiplied by the number of CPU cores used plotted against the number of CPU cores. If there were no such overhead present in CMInject, one would expect to see a horizontal line, keeping the total calculation time spent on all CPU cores constant. However, there seems to be quadratic growth in the shown curves as evident by the good fit of a polynomial of degree 2, indicating that parallelization overhead grows quadratically with the number of CPU cores \( n_{\text{CPU}} \), which becomes non-negligible at high numbers of particles and should be evaluated further. It is also notable that the overhead grows visibly more quickly when trajectories are stored and saved, which is likely due to the higher amount of data communicated between the worker processes and the main process.

The effect of this overhead becomes clearer in Figure 5.2 and Figure 5.3. The average speedup factor gained by 8 processes running on 8 virtual cores with hyperthreading, i.e., 4 physical cores, was determined to be 2.8. The average speedup factor gained by 64 processes running on 64 virtual cores, i.e., 32 physical cores, was measured to be 24.4. As the Monte Carlo nature of the problem does not in theory require any central coordination, the simulation could be distributed across multiple processes acting fully independently instead of being spawned and managed by Python’s multiprocessing module. This would also open up the possibility to distribute simulations across multiple physical machines or cluster nodes. Disk space for result storage would become the only shared resource, obviating the need for multiprocessing techniques like coordinated lock acquisition and release that typically lead to parallelization overhead, up until the point where data is written to disk or merged from multiple result files into one.

5.3. Comparisons against existing programs

Through comparison against two existing pieces of software, it is shown in this section that there has been significant progress made with regards to
5.3. Comparisons against existing programs

Figure 5.1.: The performance overhead induced by parallelization for 1000 particles. The x-axis shows \( n_{\text{CPU}} \), which is the number of CPU cores involved in the simulation, and the y-axis shows the total simulation time multiplied by \( n_{\text{CPU}} \). Fits of quadratic polynomials of all measurements are plotted as black dashed lines.

runtime performance compared to the previous version of CMInject, and that optimal performance is not yet reached compared to a Python script that was optimized for one specific kind of simulation.

5.3.1. The previous version of CMInject

As a previous version of CMInject existed, the performances of the new and old versions were compared. To this end, 3D trajectory simulations were run using both program versions with one asymmetrical 3D flow field, varying the number of particles (100 to 500, in steps of 100). To avoid statistical outliers and thus increase the confidence in the results, the same simulation was run 5 times for each program and number of particles, and initial positions and velocities were randomly generated for every run from the same distributions. The simulations were run on a “MacBook Pro (Retina, 15-inch, Late 2013)” computer with 4 physical CPU cores, once using all 8 virtual cores (\( 4 \times 2 = 8 \) cores using hyperthreading), and once using only a single process and CPU core.

The measured clock time results are shown as line plots with error bars representing standard deviation in Figure 5.2. The linear fit coefficients for the parallel runs are approximately \( f_{\text{old}}(n) \approx 0.38n + 61.83 \) and \( f_{\text{new}}(n) \approx 0.05n + 1.87 \), see Table 5.5, where \( n \) is the number of simulated parti-
5. Performance evaluation

<table>
<thead>
<tr>
<th></th>
<th>old</th>
<th>new</th>
<th>old_{st}</th>
<th>new_{st}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>0.38</td>
<td>0.05</td>
<td>0.82</td>
<td>0.13</td>
</tr>
<tr>
<td>$\beta$</td>
<td>61.83</td>
<td>1.87</td>
<td>64.64</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Table 5.5: Linear fit coefficients $\alpha$ and $\beta$, as in $y = ax + \beta$, of the runtime performance measurements shown in Figure 5.2. “st” in the subscript indicates single-threaded runs.

icles. This suggests that the new and optimized code exhibits a better time growth behavior by a factor of roughly 7.6 and has lower baseline overhead, and that runtimes of both programs grow linearly with the simulated number of particles, i.e., they exhibit $O(n)$ asymptotic runtime. Furthermore, it is visible that while speedup through parallelization has been improved from an average of 2.16 to 2.80, it does by far not reach the numbers of CPU cores. This, in addition to what has been shown in section 5.2, further indicates parallelization overhead that should be optimized.

As can also be seen in Figure 5.2, the speedup factor for parallelized simulations averages out to roughly 7.9. CMInject’s previous version exhibited a significant startup overhead, which was determined to be due to the format for the 3D flow field the program reads. This format is uncompressed and text-based, requiring the parsing of several character bytes for a single 64bit floating-point number in a 3D grid. The cminject_txt-to-hdf5 conversion tool mentioned at the beginning of chapter 4 was developed specifically to eliminate this startup overhead.

5.3.2. particleTracing.py: A team-internal tool

particleTracing.py was described in subsection 4.2.2. A performance comparison between it and CMInject is also warranted. Like in subsection 5.3.1, simulations were run varying the number of particles from 100 to 500. For each number of particles and program, a simulation was run 5 times. Simulations were performed on a computing cluster node, using Intel Xeon E5-2689 processors and 32 physical cores with hyperthreading, i.e., 64 virtual cores. As particleTracing.py only allows 2D simulations, a different simulational setup of an ALS with a long ($\approx 25$ cm) tube attached was chosen. Brownian motion was enabled, and 220 nm polystyrene particles were assumed to move through air at 293.15 K. The results are shown in Figure 5.3.
5.3. Comparisons against existing programs

Figure 5.2.: **Left**: Performance comparison showing clock time taken of the old (red, light red) and new (green, light green) version of CMInject for \( n \in \{100, 200, 300, 400, 500\} \) particles. Averaged runtimes for 5 samples each are plotted with error bars showing the sample standard deviation. Linear fits are plotted as dashed lines. “st” in the subscript indicates single-threaded runs. **Right**: Relative speedup factors after removing startup offset estimated as the y-section of each linear fit. Compared are: old vs new in parallel and single-threaded runs (blue and light blue); old code in parallel vs single-threaded (red); and new code in parallel vs single-threaded (green). Light dashed horizontal lines are the average of each plotted line.

Like in [subsection 5.3.1](#), both programs clearly exhibit \( O(n) \) runtime behavior, and the linear fit coefficients are listed in [Table 5.6](#). The average slowdown of CMInject compared to particleTracing.py for this problem is approximately 0.61 for single-threaded runs, and 0.70 for fully parallel runs. This slowdown is considered acceptable here when taking into account CMInject’s flexibility with regards to the fields, detectors, and other devices that can be defined, as well as other features introduced. It is likely that a part of this slowdown is due to the higher number of indirect function calls and generally higher dynamicism of program flow that’s naturally present in a more general framework. Neither program fully exploits parallelization, reaching only a speedup factor of 24.4 (CMInject) and 21.1 (particleTracing.py) with 32 physical (64 virtual) cores compared to single-threaded runs, see also [section 5.2](#).
5. Performance evaluation

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>p</th>
<th>C_{st}</th>
<th>p_{st}</th>
</tr>
</thead>
<tbody>
<tr>
<td>α</td>
<td>0.09</td>
<td>0.06</td>
<td>2.07</td>
<td>1.26</td>
</tr>
<tr>
<td>β</td>
<td>14.64</td>
<td>7.54</td>
<td>11.19</td>
<td>21.28</td>
</tr>
</tbody>
</table>

Table 5.6: Linear fit coefficients $\alpha$ and $\beta$, as in $y = \alpha x + \beta$, of the runtime performance measurements shown in Figure 5.3. “st” in the subscript indicates single-threaded runs. C refers to CMInject, p to particleTracing.py.

Figure 5.3: Left: Performance comparison showing clock time taken of CMInject (green, light green) and particleTracing.py (purple, light purple) version of CMInject for $n \in \{100, 200, 300, 400, 500\}$ particles. Averaged runtimes for 5 samples each are plotted with error bars showing the sample standard deviation. Linear fits are plotted as dashed lines. “st” in the subscript indicates single-threaded runs. Right: Relative speedup factors after removing startup offset estimated as the y-section of each linear fit. $p$ is particleTracing.py, C is CMInject. Compared are: $p$ vs C in parallel and single-threaded runs (blue and light blue); $p$ in parallel vs single-threaded (purple); and C in parallel vs single-threaded (green). Light dashed horizontal lines are the average of each plotted line.
5.4. Profiler output for CMInject and particleTracing.py

To make clear where most of the computation time for a typical simulation is spent, the simulation used in subsection 5.3.2 was run with both CMInject and particleTracing.py, each using only a single thread to simulate 100 particles, with Brownian motion enabled. The built-in Python profiler cProfile was used to gather statistics about the time taken by each function in each program. The output was sorted for the tottime column, which is the time taken by each function itself and not including subroutine calls. It was then reformatted and truncated to show only the functions taking at least 1% of the total runtime of each program. These outputs are listed in Appendix C. Table C.1 shows the results for CMInject, and Table C.2 shows the results for particleTracing.py.

It can be seen that CMInject has a total number of function calls roughly 1.23 as high as particleTracing.py. In both programs, the overall largest part of overall time is spent for interpolating values on a data grid (cmi
ject.utils.cython_interpolation.interp2D or fast_cubic_spline.in
terpolate_2d), on calculating an acceleration and the derivative $y'(t)$ of the ODE describing Newton’s equation of motion (Equation 2.2), and on integration of the thus numerically described problem. Another smaller part of runtime is also spent on generating random values, which is used for modeling Brownian motion.

Beyond this, CMInject has an overall larger set of functions that are called in order to do the same computational task, which together sum to a larger total runtime. Because of this, further optimizations for CMInject could perhaps be achieved by reducing subroutine calls, for instance through automatic inlining. Whether this will help reach or exceed the runtime performance of particleTracing.py is contentious, since the higher amount of abstraction present in the framework for extensibility requires a larger amount of conditionals and function calls overall.

Since the number of function calls for the overall most expensive functions is high (on the order of $10^7$), micro-optimizing these functions through benchmarking all of their primitive calls and evaluating alternatives is a worthwhile method to further reduce runtime. While it was not used for this thesis, the Python tool line_profiler, which returns profiling output for each single line in a function, could also be used here to further micro-optimize functions. Whether a primitive function takes 1 µs or 2 µs per call has a large influence on the overall runtime when such a function is
5. Performance evaluation

called $10^7$ times. CMInject’s runtimes were cut roughly in half in the last weeks of writing this thesis, through careful evaluation of profiler output and the replacement of a number of primitive calls. For example, for a Device wrapping only a single Field, the default implementation of the calculate_acceleration method for Device summed over a list of accelerations with one entry using NumPy’s `np.sum(accelerations, axis=1)`. This was found to take roughly as long as the force calculation by the Field object itself. A primitive Python loop for summing this list was found to be much faster for Field lists of $\lesssim 10$ entries. Since typical Device implementations are assumed unlikely to contain many more fields, the default implementation was then replaced with this loop. Furthermore, the implementation of calculate_acceleration for the simple FluidFlowField-Device, which wraps exactly one flow field, was replaced with an indirect call to the Field’s calculate_acceleration method, removing the need for any such loop.
A suggestion for dramatically improving performance

To make better estimations of the true particle distributions, one should be able to simulate a reasonably high number of particles while staying within an acceptable total runtime limit. To simulationally explore the parameter space for some experiment, one should be able to get usable results for any given set of parameters quickly. For both purposes, optimizing the performance of the simulation program is central. As has been shown in chapter 5, there is still work to be done here. A suggestion to possibly achieve speed improvements of orders of magnitude is described in this chapter.

6.1. Vectorization in NumPy

The NumPy library is widely used in the Python scientific computing community. Its central offering is a significant speedup (of potentially several orders of magnitude) when applying numerical operations to large datasets [39]. This speedup is possible through the NumPy array, a data structure which holds uniformly typed data arranged in a regular, potentially multidimensional pattern, and whose operations—while accessible from Python in a straightforward way—are executed in a grouped way in code written in low-level languages like C and FORTRAN. Doing so avoids the overhead of dynamic types and pointer dereferencing that comes with applying each operation to every element in a plain Python list, and is within the NumPy community referred to as “vectorization”.
6. A suggestion for dramatically improving performance

Figure 6.1.: Speedup factors from vectorized interpolation. The pale lines show the speedup for a single CPU core, and the dark lines show the speedup assuming additional parallelization of both the unvectorized and vectorized code on $\min(n_c, 64)$ CPU cores, where $n_c$ is the number of chunks. No parallelization overhead is assumed. (a) shows the speedup relative to scipy.interpolate.RegularGridInterpolator, and (b) the speedup relative to a custom 2D/3D interpolator written in Cython.

The NumPy library is already heavily in use in CMInject. The missing piece to fully vectorized performance is that particles are currently represented as Python objects and simulated one by one in parallel. A vectorizable alternative would be representing each particle as a simple array of numerical values (floating-point numbers) which are part of one larger array representing many particles at once, and parallelizing on such particle collections. The performance impact of this can be made clear by showing, for instance, how much time (bi)linear interpolation of flow field properties takes for a given number of particles in one Python list versus the particles represented as a list of multiple large arrays of numerical values, split into $n_p \in \{512, 256, 128, 64, 32, 16, 8, 4, 2\}$ chunks and vectorization applied to each piece. The results are shown in Figure 6.1.

It is visible that there is a significant speedup to be gained. For a single CPU core, the maximum achievable speedup factor is over 250 compared to the nonvectorized code. For multiple CPU cores, the maximum lies at about 175 for 50,000 particles, and is reached when the number of array chunks $n_c$ equals the number of CPU cores. The maximum achieved speedup of parallel vectorized code relative to a parallel nonvectorized implementation based on the Cython interpolator is roughly 5. The different curves in the graphs suggest that for higher numbers of particles, this factor will further improve. Assuming—very optimistically—complete orthog-
nality of the improvements gained from Cython and from NumPy vectorization, the total speed improvement of a fully vectorized pure-Cython implementation would be approximately $175 \times 5 = 875$. Assuming—rather conservatively—complete overlap of improvements by the two methods, it would be 5. The truth likely lies somewhere in the middle, but an implementation would need to be programmed and evaluated. The interpolation method the performance was compared with is trilinear interpolation. Since this method is implemented via a short series of arithmetic operations, it can be assumed that similar code would exhibit a comparable speedup.

Beyond interpolation, the force fields currently implemented in the framework would likely also benefit from being rewritten in a fashion where all parameters and returned values of functions are simple NumPy arrays rather than Python objects. This avoids frequent pointer dereferencing, potentially reduces cache misses, and allows SIMD vectorization as well as further optimizations only applicable to contiguous single-typed data to be employed. All other objects acting on particles could thus also see a significant speedup by acting on many particles at once if they were implemented in such a fashion, so that they would not become the performance bottleneck after interpolation and force calculation have been sped up by vectorization.

To evaluate this, a simplified uniform drag force without a slip correction factor was implemented both in Python and Cython, once as vectorized functions and once as simple functions to be called in a plain Python loop. These functions were then compared for between $10^3$ and $3 \times 10^7$ random particles represented as arrays storing only a 3D velocity. As parallelization overhead is negligible for force calculations repeated many times on data already communicated between processes, parallel runtimes are approximated as $1/64$ of the non-parallelized runtimes, assuming 64 CPU cores. The results are shown in Figure 6.2. Results are much more significant here, showing that a speedup of around 200 can be reached with parallelized and vectorized calculations even when using a function written in Cython. It is also shown that when comparing the parallelized and vectorized Cython implementation against a parallelized plain Python loop, the speedup factor can reach upwards of 600.
6. A suggestion for dramatically improving performance

Figure 6.2.: Runtimes and speedup factors from vectorized force calculations. (a) shows the runtimes of a plain loop, this loop’s approximated parallelized runtime on 64 cores, a vectorized implementation on a single core, and a vectorized implementation on 64 cores, separating the data into 64 individual chunks. (b) shows the speedup factors to be gained, comparing parallelized plain loops in Cython and Python, vectorized and parallelized (v&p) implementations against those plain loops in both implementations, and the overall gain of v&p Cython versus the parallelized plain Python loop.

6.2. Marrying vectorization and object-oriented code

As CMInject was written with users of varying levels of software development experience in mind, the code was designed in an object-oriented fashion to allow users to easily derive their own object implementations (see chapter 3). The question is now how to actually exploit vectorization speedups within an object-oriented framework. A solution based on metaprogramming and runtime code modification is proposed in the following.

First, particles should be represented as plain data instead of objects, preferably a single memory-contiguous array of a fixed type. Doing so is the first step to allow vectorized calculations. By defining a one-to-one correspondence between Particle instances and their array representations, all generated Particles can be converted to plain data before running a simulation, processed together as a larger array by vectorized code, and converted back to Particle instances for further processing and storage when the nu-
6.2. Marrying vectorization and object-oriented code

Numerical simulation has stopped. When more complex Particle instances containing multidimensional or multi-typed data are required, a set of particles can be represented as a collection of multiple contiguous arrays, each corresponding to one data type and shape. Defining such a mapping could either be done manually per Particle subclass, or automatically based on type-annotated data fields in the subclass definition. To enable the second part of this method to work, it is required of either method that this one-to-one mapping is not only implemented as a conversion procedure, but stored as data. For instance, a bidirectional dictionary could be employed to encode the full set of statements this mapping represents, for example

\[
\text{"particle.spatial.position} \leftrightarrow \text{array indices 0 to 2", "particle.velocity} \leftrightarrow \text{array indices 3 to 5", "particle.radius} \leftrightarrow \text{array index 6" ... .}
\]

Second, all functions and methods reading or writing particle instances should be replaced with functions that read or write multiple particle arrays at once in a vectorized fashion. The method proposed here is to use automatic code rewriting based on abstract syntax trees (ASTs). Each method operating on Particle instances is deconstructed at the start of the program into the AST of operations it represents. Each function’s Particle parameter is automatically replaced within this AST by one or multiple array parameters, depending on the complexity of the Particle subclass the function accepts. Read and write accesses on Particle instance properties within this AST are replaced with the corresponding array subscript calls. After this processing is done, the implementation of each function is swapped out before the actual simulation starts. The entire process described is possible with Python through the use of the builtin ast module and the builtin compile function.

```python
class ExampleField(Field):
    ...
    def drag_force(particle: SphericalParticle, time: float):
        pressure, velocity = self.get_local_properties(
            particle.spatial_position,
            particle.velocity
        )
        dv = velocity - particle.velocity
        return 6*pi * mu * particle.radius * dv / particle.mass
```

then automatically becomes, during program runtime:

```python
class ExampleField(Field):
    ...
```
6. A suggestion for dramatically improving performance

```python
def drag_force(particles: np.array, time: float):
    pressure, velocity = self.get_local_properties(
        particles[:,0:3],
        particles[:,3:6]
    )
    dv = velocity - particles[:,3:6]
    return 6*pi * mu * particles[:,6] * dv / particles[:,7]
```

There are further points to be considered before this technique should be implemented. One is that since numerical integration is the centerpiece of trajectory simulations as they are presented here, so an integration routine that can work on a large number of particles at once while keeping the numerical integration error—and thus, the locally chosen minimum time step—separate for each particle must be developed and included in the program. Such a routine is entirely possible and has been implemented before by Joss Wiese within the CMI group based on a 5th-order Runge-Kutta integrator, but the Python interface to the LSODA routine used in CMInject does for instance not allow the separate integration of multiple problems at once.

Another point is how the simulation of a single particle can be stopped when many particles are simulated at once. When the length of particle trajectories within a collection of simulated particles varies significantly, particles whose simulation is stopped early are nonetheless “dragged along” for simulation within the larger array representing many particles at once. When storing the full trajectories is not required, each array row that represents a particle that should not be simulated further can be hidden from further propagation or modification using an array mask, which NumPy has implemented in the `numpy.ma` module. Using such a masked array would impose an additional memory usage of $m \cdot n$ bytes, where $m$ is the number of particles and $n$ the number of data points of the longest simulated trajectory. This additional memory is $1/8$ of that of the total trajectory array; but defining better masking based only on rows and using bit arrays would decrease this to negligible $n$ bits. When trajectories, i.e., lists of particle positions and properties, should be stored in-memory and on disk, one full column of new positions is stored after each simulation step within a vectorized program. When only one particle amongst thousands takes an unusually long path, many columns containing mostly unchanged data are appended to the trajectory array, taking up a potentially large amount of memory for meaningless data. Nonetheless, this method warrants further investigation due to the possible speed improvements presented.
An extensible framework for running nanoparticle trajectory simulations, CMInject, was developed, its performance analyzed, and simulation results compared both against previously existing programs and experiments. Full documentation including a user’s guide and a developer’s guide is provided with the software. Within the CMI group, an introductory seminar on how to run and extend simulations was given. Through this, a foundation was laid to design and run many different simulations of particle trajectory experiments, using combinations of arbitrary forces, and it is now a goal of the group to further use and develop this framework.

While generally reasonable, the performance of CMInject is still lacking in some regards and should be improved. As listed in section 5.1, $10^6$ particles can be simulated on a 64 core cluster node within 1 to 2.5 hours for the asymmetric 3D problem (Table 5.1), but for simulations involving the photophoretic force (Table 5.3 and Table 5.4), only $10^4$ particles could be simulated within 8 to 16.5 hours. Furthermore, simulations with CMInject exhibit significant parallel overhead that is not inherent to the computational problem and could be avoided, see section 5.2. The Monte Carlo approach depends on a large number of particles with random initial positions to be simulated. Since the program’s performance directly affects how many particles can be simulated in some amount of time, one must accept either low result quality or long simulation times when program performance is lacking. The suggestion presented in chapter 6 for performance improvements should be considered further, and should then be implemented and tested if found to lead to the performance gains that were hoped for. Otherwise, alternative ideas should be discussed and tried. For
example, since the Monte Carlo trajectory simulation approach is highly parallelizable, it is well-suited to be solved by GPU computing, which can lead to a performance increase of orders of magnitude. Furthermore, many more supplementary tools should be added to CMInject to enable users to, for example, define and run experiments within a graphical environment; to analyze, visualize and further process result data in novel ways; or to submit and track simulations as computation cluster jobs in a straightforward manner.

The difference in particle distributions between 2D and 3D simulations shown in section 2.3 warrants further investigation. Even when using a model presented specifically for axisymmetric problems from literature [40], it can be seen that this model is not well-applicable on typical length scales the simulations were run with. This is likely due to the fact that this proposed model only adds a factor of $\sqrt{2}$ to the force’s $r$ component, assuming the force to be axisymmetric and, as such, for the orthogonal $x$ and $y$ components of acceleration and velocity to essentially act together. As all real (nano)particles move through 3D space, the Brownian force is inherently random and anything but axisymmetric. If no Brownian motion model exhibiting no significant differences between 2D and 3D simulations is developed, it should be considered to instead run all simulations in 3D and adding functionality to CMInject for explicitly defining axially symmetrical devices that do their calculations on an axially symmetrical 2D projection of the 3D particle coordinates. The toy problem runtimes showed only a negligible difference when run in 2D versus 3D (see section 5.1), so not much performance might need to be sacrificed for more accurate results with Brownian motion.

There is still much work to be done towards controlling single particles in flight. A yet hypothetical, highly efficient injection method that controls particle’s speeds, positions and injection rates in a way that increases injection efficiency would constitute a tremendous step towards atomic resolution single particle imaging, and thus towards imaging chemistry in action—taking the so-called “molecular movies”. Possible solutions to any of these problems can be approached through particle trajectory simulations—and thus with CMInject—as long as good models for all acting forces exist and can be numerically calculated in an acceptable amount of time. Using the photophoretic force for focusing has previously shown to increase hit rates for a beam of virus particles to up to 18% [14], but many more experiments for focusing with photophoresis should be conducted to evaluate its usefulness for different materials, light sources and experiment configurations. The photophoretic force still has no well-developed the-
7. Summary and outlook

Theoretical standing, especially in the Knudsen number regime of $Kn \approx 1$, where the force is most interesting since it is measured to have a maximum [26]. The model used in this thesis [32] does not account for particle absorptivities, which vary both with the particle material and the intensity profile of the laser beam. As is shown in Appendix D, properly estimating the absorption efficiency for nanoparticles requires the use of Mie theory, since nanoparticles can be close in size to the wavelength used. Furthermore, other optical parameters like the polarization of the laser light, while playing a significant effect in the magnitude of the force [41] are not considered.

Beyond already proposed mechanisms, promising new alternative approaches for controlled particle injection should be further evaluated and developed. For example, acoustic focusing as proposed in [42] can also potentially be used as a focusing mechanism. It has the advantage over photophoretic focusing that it does not rely on photon absorption, which shows a large variation between materials and is often low for biological nanoparticles at a wide range of wavelengths. Another advantage shown in [42] is that it should be possible to use acoustic methods to bunch particles together and inject them at controlled rates matching the repetition rate of an x-ray FEL laser source, further increasing the sample delivery efficiencies. Similar bunching methods could also perhaps be developed using levitation through photophoretic or other photonic forces that are activated and deactivated at controlled rates. Further possible developments of controlling particles will most likely be supported by simulations, so developing, using and improving CMInject further should be valuable to the field of single-particle imaging research.

The CMInject framework software is accessible from jochen.kuepper@cfel.de upon reasonable request.
Appendix A

CMInject’s API and documentation

A shortened and reformatted version of the base classes’ Application Programming Interfaces (APIs) described in section 3.2 is listed below.

The full documentation, which includes a User Guide and a Developer Guide along with the docstrings and APIs of all classes and methods in CMInject, is available with the program. It can be generated by running `make html` inside the `doc/` subdirectory, and can then be viewed in a browser by opening the file `doc/_build/html/index.html`.

```python
from abc import ABC, abstractmethod
from typing import Tuple, List, Any, Optional
import numpy as np

class NDimensional(ABC):
    
    """
    A mixin for objects that are N-dimensional, or at least (may) act differently depending on the number of spatial dimensions of the simulation setup they are used within.
    """
    @abstractmethod
    def set_number_of_dimensions(self, number_of_dimensions: int):
        """
        A method to change this object’s data/state/implementation/... to conform to the choice of the simulation setup’s number of spatial dimensions.
        """

class ZBounded(ABC):
```
A. CMInject’s API and documentation

"""
A mixin for objects in an experiment setup that are bounded in
the Z direction. Classes deriving from this mixin will have to
implement the get_z_boundary method.
"""
@property
@abstractmethod
def z_boundary(self) -> Tuple[float, float]:
    """Returns the Z boundary of this Z-bounded object."""

class Particle(NDimensional, ABC):
    """
    Describes a particle whose trajectory we want to simulate.
    It is first and foremost a data container, and it and its subclasses
    should be written and used as such.
    """
def __init__(self, identifier: Any, start_time: float,
            position: np.array, *args, **kwargs):
    ...

@property
@abstractmethod
def properties(self) -> np.array:
    """
    A (n,)-dimensional numpy array of all properties - beyond the
    usual phase space position (position+velocity) - that describe
    the particle's current state in a way that is useful to the
    problem domain.
    """

@property
@abstractmethod
def properties_description(self) -> List[str]:
    """
    A list of strings matching the .properties attribute in length,
    describing each value in the array in some manner (most likely
    using standard physical abbreviations like rho, phi, T, k, ...).
    """

@property
def position_description(self) -> List[str]:
    """
    A list of strings matching the .position attribute in length,
    describing each value in the array in some manner (most likely
    using standard physical abbreviations like x, y, z, vx, vy, vz, ...).
    """

class ParticleDetectorHit(object):

54
A small data container class, to store useful information about a hit of a particle on a detector.

def __init__(self, hit_position: np.array, particle: Particle):
...

@property
def hit_position(self) -> np.array:
    """The position where the detector detected the particle hit."""

class Source(NDimensional, ABC):
    """
    A source of particles, to generate an initial position (and property) distribution.
    """
    @abstractmethod
def generate_particles(self, start_time: float = 0.0) -> List[Particle]:
        """
        Generates a list of particles. How this is done is entirely up to the subclass, by (this is mandatory!) implementing this method in some way.
        """

class Detector(NDimensional, ZBounded, ABC):
    """
    Can tell whether a Particle has hit it, and _if_ it has hit it, can also tell where this occurred.
    """
def __init__(self, identifier: int):
...

def try_to_detect(self, particle: Particle) -> bool:
    """
    Tries to detect a particle based on the _has_particle_reached_detector and _hit_position methods.
    """
    @abstractmethod
def _has_particle_reached_detector(self, particle_identifier: int, position_velocity: np.array) -> bool:
        """
        Tells whether a Particle has reached this Detector. Must return True/False.
        """
    @abstractmethod
def _hit_position(self, position_velocity: np.array) -> Optional[np.array]:

A. CMInject’s API and documentation

Can return a hit position on this detector for a Particle, but might also return None (if the Particle isn’t considered to have reached this detector).

class Boundary(NDimensional, ZBounded, ABC):
    """
    Can tell whether a Particle is inside of it or not.
    """
    @abstractmethod
def is_particle_inside(self, position: np.array, time: float) -> bool:
        """
        Tells whether the passed Particle is inside of this Boundary or not.
        """

class Field(NDimensional, ZBounded, ABC):
    """
    An acceleration field interacting with Particle objects.
    """
    @abstractmethod
def calculate_acceleration(self, particle: Particle, time: float) -> np.array:
        """
        Calculates an acceleration for one particle based on the particle’s current properties and the current time. This acceleration will be integrated for in each time step and thus "applied" to the particle.
        """

class Device(NDimensional, ZBounded, ABC):
    """
    A combination of Field instances and a Boundary instance. Used to model real-world devices in an experiment setup.
    """
    def __init__(self, fields: List[Field], boundary: Boundary):
        ...

class PropertyUpdater(NDimensional, ABC):
    """
    An object to update a Particle’s properties based on its current state. Can do arbitrary calculations to determine the values the properties should be set to.
    """
    @abstractmethod
def update(self, particle: Particle, time: float) -> bool:
        """
        Updates a property of some Particle instance in some way.
        """

class ResultStorage(ABC):
    """
An object to store the results of an experiment in some fashion. MUST implement store_results, and MAY implement convenience methods to read from the storage again (e.g. a method to get all particle trajectories from a file).

```python
@abstractmethod
def store_results(self, particles: List[Particle]):
    
    Stores the results of an experiment (which are always a list of modified Particle instances).
```

Appendix B

Code for the simulated multiphysics setup

The code defining the multiphysics setup described in section 4.4 is listed below. It is located in the cminject.definitions.setups.gold adl module.

```python
import argparse
from typing import Tuple
import numpy as np

from cminject.experiment import Experiment
from cminject.utils.args import SetupArgumentParser
from cminject.definitions import Device, Boundary
from cminject.definitions.setups import Setup

from cminject.definitions.boundaries import GridFieldBasedBoundary
from cminject.definitions.detectors import SimpleZDetector
from cminject.definitions.devices.desyatnikov_photophoresis_device import DesyatnikovPhotophoresisDevice
from cminject.definitions.fields.fluid_flow_fields import StokesDragForceField
from cminject.definitions.particles import ThermallyConductiveSphericalParticle
from cminject.definitions.property_updaters import BrownianMotionPropertyUpdater
from cminject.definitions.sources import VariableDistributionSource

class SkimmersBoundary(Boundary):
    def __init__(self, first_skimmer_flow_field: StokesDragForceField, inter_distance: float, skimmer_length: float, tube_length: float,
```
B. Code for the simulated multiphysics setup

```python
skimmer_min_radius: float, skimmer_max_radius: float,
skimmer_min_z: float):
self.first_skimmer_flow_field = first_skimmer_flow_field
self.inter_distance = inter_distance
self.skimmer_length = skimmer_length
self.tube_length = tube_length
self.skimmer_min_radius = skimmer_min_radius
self.skimmer_max_radius = skimmer_max_radius

self._z_boundary = (skimmer_min_z, skimmer_length + tube_length)

def set_number_of_dimensions(self, number_of_dimensions: int):
    if number_of_dimensions != 2:
        raise ValueError("The number of dimensions shall be 2!")

def is_particle_inside(self, position: np.array, time: float) -> bool:
    r, z = position
    r = np.abs(r)
    zmin, zmax = self.z_boundary
    if not (zmin <= z <= zmax):
        return False

    if z <= 0.0:
        return self.first_skimmer_flow_field.
           .is_particle_inside(position, time)
    elif z <= self.inter_distance:
        # In between we never consider particles to be lost
        return True
    elif self.inter_distance < z <= self.skimmer_length:
        # In the trapezoid, verify the particle is within
        # the offset triangle
        lng = self.skimmer_length
        r0, r1 = self.skimmer_min_radius, self.skimmer_max_radius
        return z >= lng*((r-r0) / (r1-r0))
    else:
        # In the rest of the Z range, verify the particle is
        # inside the tube
        return r <= self.skimmer_max_radius

@property
def z_boundary(self) -> Tuple[float, float]:
    return self._z_boundary

NITROGEN_PARAMS = {
    'dynamic_viscosity': 1.76e-5,  # [Pa*s]
    'm_gas': 4.7e-26,  # [kg]
}```
B. Code for the simulated multiphysics setup

```python
'temperature': 293.15,  # [K]
'slip_correction_model': 'room_temp'
}

class SkimmersDevice(Device):
    def __init__(self, filename, inter_distance: float,
                 skimmer_length: float, tube_length: float,
                 skimmer_min_r: float, skimmer_max_r: float,
                 skimmer_min_z: float):
        field = StokesDragForceField(filename=filename, **NITROGEN_PARAMS)
        boundary = SkimmersBoundary(
            first_skimmer_flow_field=field,
            inter_distance=inter_distance,
            skimmer_length=skimmer_length,
            tube_length=tube_length,
            skimmer_min_radius=skimmer_min_r,
            skimmer_max_radius=skimmer_max_r,
            skimmer_min_z=skimmer_min_z)

        super().__init__(fields=[field], boundary=boundary)

class ADLStackDevice(Device):
    def __init__(self, filename, z_offset):
        field = StokesDragForceField(filename=filename, **NITROGEN_PARAMS,
                                   offset=np.array([0, z_offset]))
        boundary = GridFieldBasedBoundary(field)
        super().__init__(fields=[field], boundary=boundary)

class GoldADLSetup(Setup):  
    @staticmethod
    def construct_experiment(main_args: argparse.Namespace, 
                              args: argparse.Namespace) -> Experiment:
        skimmer_flow_file = args.skimmer1_file
        adl_flow_file = args.adl_file

        inter_distance = 2.95e-3
        skimmer_length = 2e-2
        skimmer_tube_length = 15e-2
        skimmer_min_z = -2.35835e-2
        skimmer_min_r = 0.5e-3
        skimmer_max_r = 2.0e-2

        exit_offset = 0.0003
        adl_extent = 0.4272
```
# B. Code for the simulated multiphysics setup

\[ \text{wtf\_offset} = -1e^{-3} \]
\[ \text{adl\_offset} = \text{exit\_offset} + \text{adl\_extent} + \text{wtf\_offset} + \]
\[ \quad \text{skimmer\_length} + \text{inter\_distance} + \text{skimmer\_tube\_length} \]

# 0.60015 is where the entire ADL flow field terminates.
# this is a bit after the exit, the exit position is at .59915
\[ \text{adl\_exit\_position} = 0.59915 \]

\text{devices} = [\]
\[ \quad \text{SkimmersDevice(} \]
\[ \quad \quad \text{skimmer\_flow\_file, inter\_distance=inter\_distance,} \]
\[ \quad \quad \text{skimmer\_length=skimmer\_length,} \]
\[ \quad \quad \text{tube\_length=skimmer\_tube\_length,} \]
\[ \quad \quad \text{skimmer\_min\_r=skimmer\_min\_r, skimmer\_max\_r=skimmer\_max\_r,} \]
\[ \quad \quad \text{skimmer\_min\_z=skimmer\_min\_z} \]
\[ \quad \),} \]
\[ \quad \text{ADLStackDevice(adl\_flow\_file, z\_offset=adl\_offset),} \]
\]
\# only need to create a device if beam\_power is set
\if args.beam\_power is not None:
\[ \quad \text{devices} += [\]
\[ \quad \quad \text{DesyatnikovPhotophoresisDevice(} \]
\[ \quad \quad \quad \text{gas\_density=(0.059, devices[1].fields[0]),} \]
\[ \quad \quad \quad \text{gas\_viscosity=17.8771e-6,} \quad \text{[Pa\_s]} \]
\[ \quad \quad \quad \text{gas\_thermal\_conductivity=25.9361e-3,} \quad \text{[W/(m\_K)]} \]
\[ \quad \quad \quad \text{gas\_mass=4.652e-26,} \quad \text{[kg], one molecule} \]
\[ \quad \quad \quad \text{gas\_temperature=293.15,} \quad \text{[K]} \]
\[ \quad \quad \quad \text{beam\_waist\_radius=args.beam\_waist\_radius,} \quad \text{[m]} \]
\[ \quad \quad \quad \text{beam\_power=args.beam\_power,} \quad \text{[W]} \]
\[ \quad \quad \quad \text{r\_boundary=(-1e^{-3}, 1e^{-3}),} \quad \text{[m]} \]
\[ \quad \quad \quad \text{z\_boundary=(adl\_exit\_position, adl\_exit\_position + 1e^{-1}),} \]
\[ \quad \quad \quad \text{z\_position=adl\_exit\_position + 1e^{-2},} \quad \text{[m]} \]
\[ \quad \)] \]
\]
\text{detectors} = [\]
\[ \quad \text{SimpleZDetector(i, adl\_exit\_position + i*5e^{-4})} \]
\[ \quad \text{for i in range(20)} \]
\]
\text{sources} = [\]
\[ \quad \text{VariableDistributionSource(} \]
\[ \quad \quad \text{subclass=ThermallyConductiveSphericalParticle,} \]
\[ \quad \quad \text{number\_of\_particles=main\_args.nof\_particles,} \]
\[ \quad \quad \text{rho=19320.0,} \quad \text{[Assuming 50nm gold particles]} \]
\[ \quad \quad \text{radius=50e^{-9},} \]
\]

62
**B. Code for the simulated multiphysics setup**

```python
thermal_conductivity=315.0,  # [W / (m*K)] for Au
specific_heat=0.0,
temperature=293.15,
position=[[{'kind': 'radial_gaussian',
            'mu': 0.0, 'sigma': 3.0e-3},
           skimmer_min_z+abs(skimmer_min_z*0.001)],
       velocity=[[{'kind': 'gaussian',
                     'mu': 1e-3, 'sigma': 1e-5},
                  {'kind': 'gaussian',
                   'mu': 0.155, 'sigma': 0.001}]
]

# Construct the Experiment, still missing brownian motion
exp = Experiment(
    detectors=detectors,
    sources=sources,
    devices=devices,
    time_step=1e-5,
    number_of_dimensions=2
)

if args.brownian:
    # Add brownian motion property updaters _after_ the Experiment
    # has decided on a time step to use.
    dt = exp.time_interval[-1]
    bm_updaters = [
        BrownianMotionPropertyUpdater(field=d.fields[0], dt=dt)
        for d in devices[[0, 1]]
    ]
    for updater in bm_updaters:
        updater.set_number_of_dimensions(exp.number_of_dimensions)
    exp.property_updaters = bm_updaters + exp.property_updaters

return exp

@staticmethod
def get_parser() -> SetupArgumentParser:
    parser = SetupArgumentParser()

    parser.add_argument(
        '-fS1', '--skimmer1-file',
        help='The flow field file (HDF5) for the first skimmer',
        required=True, type=str
    )
    parser.add_argument(
        '-fADL', '--adl-file',
        help='The flow field file (HDF5) for the ADL setup',
        required=True, type=str
    )
```

---

63
B. Code for the simulated multiphysics setup

Listing 1: The code defining the example multiphysics setup from section 4.4.

```python
import argparse

def parser():
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '-bP', '--beam-power',
        help='Beam power of a photophoretic LG01 vortex laser beam [W]',
        type=float
    )
    parser.add_argument(
        '-bw0', '--beam-waist-radius',
        help='Beam waist radius of a photophoretic LG01 vortex laser beam [m]',
        type=float, default=8e-6
    )
    parser.add_argument(
        '-B', '--brownian',
        help='Enable brownian motion',
        action='store_true'
    )
    return parser
```
Appendix C

Profiler output for CMInject and particleTracing.py

Formatted and truncated profiler output for single-threaded runs of CMInject and particleTracing.py, both simulating the same problem with 100 particles, is listed in the following. $t_{\text{tot}}$ corresponds to $\text{tottime}$ column in cProfile output, which is the time each function took, excluding subprocedure calls. $t_{\text{cum}}$ corresponds to $\text{cumtime}$, which is the same as $\text{tottime}$ but including subprocedure calls. $t_{\text{tot,p}}$ and $t_{\text{cum,p}}$ refer to the same measurements respectively, but an average of the time per function call instead of the sum of all calls. $t_{\text{tot}}/t$ is the fraction of $t_{\text{tot}}$ of the total program runtime $t$, and $\text{cum}(t_{\text{tot}}/t)$ is the cumulative sum of $t_{\text{tot}}$. The rows are sorted based on the $t_{\text{tot}}$ column, and functions taking
C. Profiler output for CMInject and particleTracing.py

CMInject

174609612 function calls (174555177 primitive calls) in 371.329 seconds

<table>
<thead>
<tr>
<th>Calls</th>
<th>$t_{tot}$</th>
<th>$t_{tot,p}$</th>
<th>$t_{cum}$</th>
<th>$t_{cum,p}$</th>
<th>$t_{tot}/t$</th>
<th>cum($t_{tot}/t$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{built-in method cminject.utils.cython_interpolation.interp2D}</td>
<td>1.42 × 10^7</td>
<td>43.623</td>
<td>0.000</td>
<td>0.000</td>
<td>11.75%</td>
<td>11.75%</td>
</tr>
<tr>
<td>&lt;C&gt;/experiment.py:32(spatial_derivatives)</td>
<td>6.52 × 10^6</td>
<td>39.244</td>
<td>0.000</td>
<td>272.019</td>
<td>10.57%</td>
<td>22.32%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:120(calculate_acceleration)</td>
<td>6.52 × 10^6</td>
<td>33.060</td>
<td>0.000</td>
<td>133.555</td>
<td>8.90%</td>
<td>31.22%</td>
</tr>
<tr>
<td>&lt;C&gt;/utils/interpolation.py:32(<em>call</em>)</td>
<td>1.42 × 10^7</td>
<td>31.232</td>
<td>0.000</td>
<td>83.661</td>
<td>8.41%</td>
<td>39.63%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:173(_calc_slip_correction_Hutchins)</td>
<td>7.12 × 10^6</td>
<td>25.111</td>
<td>0.000</td>
<td>28.535</td>
<td>6.76%</td>
<td>46.39%</td>
</tr>
<tr>
<td>&lt;SP&gt;/scipy/integrate/ode.py:1335(run)</td>
<td>6.04 × 10^5</td>
<td>23.416</td>
<td>0.000</td>
<td>296.440</td>
<td>6.31%</td>
<td>52.70%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:44(get_local_properties)</td>
<td>6.52 × 10^6</td>
<td>17.324</td>
<td>0.000</td>
<td>56.017</td>
<td>4.67%</td>
<td>57.36%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/property_updaters.py:66(update)</td>
<td>6.04 × 10^5</td>
<td>15.269</td>
<td>0.000</td>
<td>27.652</td>
<td>4.11%</td>
<td>61.48%</td>
</tr>
<tr>
<td>{built-in method numpy.concatenate}</td>
<td>7.12 × 10^6</td>
<td>14.876</td>
<td>0.000</td>
<td>14.876</td>
<td>4.01%</td>
<td>65.48%</td>
</tr>
<tr>
<td>{built-in method numpy.zeros}</td>
<td>7.73 × 10^6</td>
<td>13.752</td>
<td>0.000</td>
<td>13.752</td>
<td>3.70%</td>
<td>69.19%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:51(interpolate)</td>
<td>1.42 × 10^7</td>
<td>12.721</td>
<td>0.000</td>
<td>96.388</td>
<td>3.43%</td>
<td>72.61%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:113(_a)</td>
<td>6.52 × 10^6</td>
<td>11.876</td>
<td>0.000</td>
<td>11.876</td>
<td>3.20%</td>
<td>78.61%</td>
</tr>
<tr>
<td>&lt;C&gt;/definitions/fields/fluid_flow_fields.py:57(is_particle_inside)</td>
<td>7.12 × 10^6</td>
<td>11.694</td>
<td>0.000</td>
<td>65.626</td>
<td>3.15%</td>
<td>78.96%</td>
</tr>
<tr>
<td>{method ‘copy’ of ‘numpy.ndarray’ objects}</td>
<td>1.42 × 10^7</td>
<td>8.812</td>
<td>0.000</td>
<td>8.812</td>
<td>2.37%</td>
<td>81.33%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

66
C. Profiler output for CMInject and particleTracing.py

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Calls</th>
<th>$t_{tot}$</th>
<th>$t_{tot,p}$</th>
<th>$t_{cum}$</th>
<th>$t_{cum,p}$</th>
<th>$t_{tot}/t$</th>
<th>cum($t_{tot}/t$)</th>
</tr>
</thead>
</table>
| {fast_cubic_spline.interpolate.
| $2.10 \times 10^7$ | 43.011 | 0.000 | 43.011 | 0.000 | 20.24% | 20.24% |
| {method 'normal' of 'mtrand.RandomState' objects} | | | | | | | |
| $6.04 \times 10^5$ | 5.266 | 0.000 | 5.266 | 0.000 | 1.42% | 88.42% |
| {method 'normal' of 'mtrand.RandomState' objects} | | | | | | | |
| $6.05 \times 10^5$ | 4.549 | 0.000 | 5.919 | 0.000 | 1.23% | 90.87% |
| {method 'normal' of 'mtrand.RandomState' objects} | | | | | | | |
| $7.12 \times 10^6$ | 4.322 | 0.000 | 69.949 | 0.000 | 1.16% | 92.04% |
| {method 'normal' of 'mtrand.RandomState' objects} | | | | | | | |
| $7.12 \times 10^6$ | 3.932 | 0.000 | 3.932 | 0.000 | 1.06% | 93.10% |

Table C.1.: Truncated cProfile output for CMInject running 100 particles single-threaded, sorted by internal time of each function. <C> is a placeholder for CMInject's lib/ directory, and <SP> is a placeholder for the site-packages directory of the Python 3.7 installation used. The columns are explained at the beginning of this appendix section.

**particleTracing.py**

142172000 function calls (141525460 primitive calls) in 212.541 seconds
C. Profiler output for CMInject and particleTracing.py

\[
\begin{array}{cccccc}
\text{function} & \text{time (ms)} & \text{count} & \text{duration (ms)} & \text{percent} & \text{total percent} \\
\text{<P>:592(derivative)} & 6.79 \times 10^6 & 19.942 & 0.000 & 103.264 & 0.000 & 9.38\% & 53.85\% \\
\text{<P>:601(fly)} & 1.00 \times 10^2 & 14.843 & 0.148 & 205.694 & 2.057 & 6.98\% & 60.84\% \\
\{\text{method 'normal' of 'numpy.random.mtrand.RandomState' objects}\} & 1.28 \times 10^6 & 10.837 & 0.000 & 10.837 & 0.000 & 5.10\% & 65.94\% \\
\{\text{built-in method numpy.array}\} & 9.35 \times 10^6 & 9.736 & 0.000 & 9.736 & 0.000 & 4.58\% & 70.52\% \\
\text{<SP>/scipy/interpolate/interpolate.py:671(_check_bounds)} & 6.39 \times 10^3 & 5.643 & 0.000 & 5.643 & 0.000 & 2.66\% & 73.17\% \\
\text{<SP>/scipy/integrate/ode.py:1285(reset)} & 6.40 \times 10^5 & 5.273 & 0.000 & 6.755 & 0.000 & 2.48\% & 75.65\% \\
\text{<SP>/scipy/lib/util.py:193(_asarray_validated)} & 6.40 \times 10^5 & 3.914 & 0.000 & 9.396 & 0.000 & 1.84\% & 77.49\% \\
\{\text{cython_functions.Cc}\} & 1.49 \times 10^7 & 3.149 & 0.000 & 3.149 & 0.000 & 1.48\% & 78.97\% \\
\{\text{cython_functions.Fdrag}\} & 1.36 \times 10^7 & 2.852 & 0.000 & 2.852 & 0.000 & 1.34\% & 80.32\% \\
\{\text{cython_functions.Kn}\} & 1.49 \times 10^7 & 2.794 & 0.000 & 2.794 & 0.000 & 1.31\% & 81.63\% \\
\text{<SP>/scipy/integrate/ode.py:359(set_initial_value)} & 6.40 \times 10^5 & 2.734 & 0.000 & 16.827 & 0.000 & 1.29\% & 82.92\% \\
\text{<SP>/scipy/interpolate/interpolate.py:656(_evaluate)} & 6.39 \times 10^3 & 2.279 & 0.000 & 15.579 & 0.000 & 1.07\% & 83.99\% \\
\end{array}
\]

Table C.2.: Truncated cProfile output for particleTracing.py running 100 particles single-threaded, sorted by internal time of each function. <P> is a placeholder for particleTracing.py, and <SP> is a placeholder for the site-packages directory of the Python 3.7 installation used. The columns are explained at the beginning of this appendix section.
In this chapter, a comparison between two approximations, one based on Fresnel equations and one on Mie theory, for a measure of absorption efficiency $\varepsilon \in [0,1]$ is made after the necessary equations and computations are defined.

**Fresnel equations**

To approximate $\varepsilon$ with Fresnel equations I first define, for a single incoming light wave, angle-dependent reflectivity components $R_{s,p}$, transmissivity components $T_{s,p}$, and absorption components $A_{s,p}$. $p$ is the component parallel to an incident lightwave, and $s$ is the component orthogonal to it. The reflectivity coefficients are defined as follows [26]:

\[
R_s(\kappa_1, \kappa_2, \theta_i) = \frac{|\kappa_1 \cos(\theta_i) - \kappa_2 \cos(\theta_r)|^2}{|\kappa_1 \cos(\theta_i) + \kappa_2 \cos(\theta_r)|^2} \tag{D.1}
\]

\[
R_p(\kappa_1, \kappa_2, \theta_i) = \frac{|\kappa_1 \cos(\theta_r) - \kappa_2 \cos(\theta_i)|^2}{|\kappa_1 \cos(\theta_r) + \kappa_2 \cos(\theta_i)|^2} \tag{D.2}
\]

where $\kappa_{1,2}$ are the complex refractive indices (CRIs) of the materials involved, with light traveling from the material with $\kappa_1$ to the one with $\kappa_2$. $\theta_i$ is the incident angle, and $\theta_r = \arcsin \frac{\kappa_1}{\kappa_2} \sin(\theta_i)$ the refracted angle according to Snell’s law.

Defining the absorption components $A_{s,p}$ as $A_{s,p} = 1 - R_{s,p}$ would be a reasonable approximation for materials that are fully or almost fully
opaque at a certain wavelength, like metals or graphite are to most visible light: In this case, all light not reflected can be assumed to have been absorbed. For materials which are mostly transparent to a given wavelength, this assumption does not hold. Instead, based on the law of exponential attenuation of light through a material with a nonzero attenuation coefficient \cite[p. 219]{43}—which is the same as $k$, the imaginary part of the CRI—I define additional transmission coefficients $T_s$ and $T_p$. These are dependent not only on the CRIs and incident angle, but also on $\lambda$ and on $r$, which is the particle radius. $r|\cos(\theta_r)|$ is taken to be the length the attenuated light wave travels, at maximum, through the spherical particle.

\begin{align}
T_s(\kappa_1, \kappa_2, \theta_i, r, \lambda) &= (1 - R_s(\kappa_1, \kappa_2, \theta_i)) \exp\left(\frac{-2\pi k_2 \cdot r|\cos(\theta_r)|}{\lambda}\right) \quad (D.3) \\
T_p(\kappa_1, \kappa_2, \theta_i, r, \lambda) &= (1 - R_p(\kappa_1, \kappa_2, \theta_i)) \exp\left(\frac{-2\pi k_2 \cdot r|\cos(\theta_r)|}{\lambda}\right) \quad (D.4)
\end{align}

Based on both $R_s, p$ and $T_s, p$, an approximation for the absorption components that makes this model applicable to transparent materials can be defined:

\begin{align}
A_s(\kappa_1, \kappa_2, \theta_i, r, \lambda) &= 1 - R_s(\kappa_1, \kappa_2, \theta_i) - T_s(\kappa_1, \kappa_2, \theta_i, r, \lambda) \quad (D.5) \\
A_p(\kappa_1, \kappa_2, \theta_i, r, \lambda) &= 1 - R_p(\kappa_1, \kappa_2, \theta_i) - T_p(\kappa_1, \kappa_2, \theta_i, r, \lambda) \quad (D.6)
\end{align}

I then define $\epsilon_F$ in terms of a normalized integral over a half-sphere surface, on which all light is uniformly incident:

\begin{align}
\epsilon_F(\kappa_1, \kappa_2, r, \lambda) := \frac{1}{2\pi} \int_0^{\pi/2} \int_0^{2\pi} \eta(\kappa_1, \kappa_2, \theta, r, \lambda) \cos(\theta) \sin(\theta) \, d\phi \, d\theta \\
\eta := \sin(\phi)^2 A_s + \cos(\phi)^2 A_p \quad (D.7)
\end{align}

**Mie theory**

Mie’s solution to Maxwell’s equations \cite{44} is exact for homogenous spheres, but computable closed forms are not known, so results must be computed numerically. Neither the details of the Mie solution nor of its numerical computation are investigated or discussed here. The Python package miepython written specifically for this purpose \cite{45} was imported, and the code listed in \textbf{Listing 2} used to retrieve $\epsilon_M := \frac{Q_{\text{abs}}}{Q_{\text{ext}}} = \frac{Q_{\text{ext}} - Q_{\text{sca}}}{Q_{\text{ext}}}$. $Q_{\text{ext}}$ is the extinction cross section, $Q_{\text{sca}}$ the scattering cross section, and $Q_{\text{abs}} = Q_{\text{ext}} - Q_{\text{sca}}$ the absorption cross section.
import numpy as np
import miepython as mp

def epsilon_M(kappa, r, lambda_):
    x = 2 * np.pi * r * lambda_
    qext, qsc, _, _ = mp.mie(kappa, x)
    qabs = qext - qsc
    return qabs / qext

Listing 2: The code for retrieving $\epsilon_M$ with miepython.

Comparison of Fresnel equations and Mie theory

The materials gold (Au), copper (Cu), graphite (C, amorphous) and polystyrene ([$C_8H_8$]$_n$) are compared for both approximations. A fixed light wavelength of $\lambda_1 = 532$ nm is chosen since it is emitted by typical Nd:YAG laser sources. Gold and copper were chosen since they are opaque and somewhat reflective at this wavelength, graphite as it is highly opaque and nonreflective, and polystyrene as it is almost fully transparent. For polystyrene, a wavelength of $\lambda_2 = 1.4$ µm was evaluated as well since the material is less transparent to it. $\kappa_1$ is set to be 1, as the medium the light wave was incident from is assumed to be air. The CRIs $\kappa$ of the considered materials are listed in Table D.1. The results and their comparison are shown in Figure D.1 and the comparison in the limit of large particle sizes in Figure D.2. The two approaches are compared by dividing $\epsilon_F / \epsilon_M$, as $\epsilon_F$ is below $\epsilon_M$ for all materials and particle sizes. It can be seen that:

1. The Fresnel approximation $\epsilon_F$ can severely underestimate the absorption efficiency for particles with a radius close to the light wavelength, yielding absorption efficiencies as low as $\approx 5\%$ of those by Mie theory, see Figure D.1.

2. This estimation error is especially high for particles consisting of transparent materials, where it also varies quickly with small changes in particle radius, see Figure D.1 (b) and (c).

3. In the limit of $r \gg \lambda$, where $\lambda$ is the light wavelength and $r$ is the particle radius, the Fresnel approximation converges to the Mie solution up to numerical error, see Figure D.2.
D. Absorption efficiencies, Fresnel versus Mie

Figure D.1.: A comparison for an absorption efficiency $\varepsilon$ based on Fresnel equations $\varepsilon_F$ and numerical results from Mie theory $\varepsilon_M$, for various materials listed in Table D.1. The relationship between radius and wavelength $r/\lambda$ was varied from 0.01 to 10. (a) shows the total calculated absorption efficiencies, $\varepsilon_M$ shown in darker colors and $\varepsilon_F$ in lighter colors. (b) shows their quotient $\varepsilon_F/\varepsilon_M$ and marks its minima with dotted lines, i.e., the maximum error of underestimation by the Fresnel approximation. Since all quotients reach zero as $r/\lambda$ tends to 0, the gray hatched region was ignored when determining those minima. (c) shows a subregion of the plot in (b), for easier visualization of how $\varepsilon_F/\varepsilon_M$ changes for small changes in $r$ when a highly transparent material like Polystyrene is used.
D. Absorption efficiencies, Fresnel versus Mie

<table>
<thead>
<tr>
<th>Material</th>
<th>$\lambda$</th>
<th>$n$</th>
<th>$k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Au</td>
<td>532 nm</td>
<td>0.54386</td>
<td>2.2309</td>
</tr>
<tr>
<td>Cu</td>
<td>532 nm</td>
<td>1.1159</td>
<td>2.5956</td>
</tr>
<tr>
<td>Graphite</td>
<td>532 nm</td>
<td>2.33</td>
<td>1.33</td>
</tr>
<tr>
<td>$[C_8H_8]_n$</td>
<td>532 nm</td>
<td>1.595</td>
<td>0.0005</td>
</tr>
<tr>
<td>$[C_8H_8]_n$</td>
<td>1.4 µm</td>
<td>1.572</td>
<td>0.015</td>
</tr>
</tbody>
</table>

Table D.1.: The complex refractive indices (CRIs) $\kappa = n + ik$ of various materials at fixed wavelengths.

Figure D.2.: The absorption efficiencies $\varepsilon_{F,M}$ as in Figure D.1 but with the relationship between radius and wavelength $r/\lambda$ varied from 5 to 10$^3$, illustrating the convergence of the Fresnel approximations towards the numerical Mie results for large particle sizes. As in Figure D.1 (a) shows the total calculated absorption efficiencies, $\varepsilon_M$ drawn in darker colors and $\varepsilon_F$ in lighter colors, and (b) shows their quotient $\varepsilon_F/\varepsilon_M$. 

73
Bibliography


Bibliography


Eidesstattliche Versicherung


Hamburg, den 07.10.2019

________________________
Simon Welker

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 07.10.2019

________________________
Simon Welker