2023

ANALYSIS OF A NANOPARTICLE’S DYNAMICS IN AN OPTICAL TWEEZER

Vatsal Asitkumar Joshi

ANALYSIS OF A NANOPARTICLE'S DYNAMICS IN AN OPTICAL TWEEZER

by

VATSAL ASITKUMAR JOSHI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY
in
MECHANICAL ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON
December 2023

Supervising Committee:

Dr. Alan Bowling (Committee Chairperson)

Dr. Hyejin Moon

Dr. Ashfaq Adnan

Dr. Albert Tong

Dr. Georgios Alexandrakis

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Alan Bowling for constantly motivating and encouraging me. His guidance has helped me during the time of this research. I would also like to thank Dr. Hyejin Moon, Dr. Ashfaq Adnan, Dr. Georgios Alexandrakis and Dr. Albert Tong for being part of my defense committee and providing their time and expertise in reviewing my work.

I would like to thank Dr. Peter Goodwin from the Los Alamos National Laboratory, Dr. Walter Paxton from the Brigham Young University and Dr. Alexander Stilgoe from the University of Queensland for providing valuable experimental data and guidance on building the experimental setup. I am extremely grateful to Dr. Michael Cho for lending the equipment and providing me necessary guidance for the instruments used. I would also like to thank Dr. Samarendra Mohanty and Dr. Mahdi Haghshenas-Jaryani for the numerous discussions regarding optical tweezers.

This work would not have been possible without the support of my current and previous lab members, Dr. Ashley Guy, Dr. Abhishek Chatterjee, Manoochehr Rabiei, Regan Kubicek, Eric Havenhill, Kangji Huang and Cristian Almendariz at the Biomechanics Robotics and Dynamic Systems Laboratory. Thank you for making the workplace feel like a second home to me.

I am grateful to all the teachers in India and here in the United States of America, especially to Dr. David Hullender, Dr. Kamesh Subbarao and Dr. Jason Losh for igniting my interest in the fields of dynamics, controls and embedded systems. I am also very grateful to my high school teacher Mr. Santosh Kaushik for making me good at self-study, which has been very useful in my research.

Finally, I would like to express my deepest gratitude to my parents and my sister who have always encouraged me and have always been supportive. I would also like to thank all of my friends who have been there for me through thick and thin.

December 14, 2023

ABSTRACT

ANALYSIS OF A NANOPARTICLE'S DYNAMICS
IN AN OPTICAL TWEEZER

Vatsal Asitkumar Joshi, Ph.D.
The University of Texas at Arlington, 2023

Supervising Professor: Dr. Alan Bowling

This work presents a simulation and experimental analysis of a nanoparticle's motion in an optical tweezer. Specifically, different simulation and analysis techniques are developed to investigate a suspected regime change in the dynamics of the micro/nano particles from overdamped to underdamped motion as the particle size reduces to submicron scale. Moreover, the simulation techniques developed here provide accurate prediction of the particle dynamics along with low computation time. Finally, the experimental setup developed here provides new experimental data that help answer the question of the regime change with increased certainty.

Micro- and nano- scale systems, such as an optical tweezer, require a longer time to simulate because of the disproportionality that exists between fluid and inertia forces. This problem is resolved by the use of a multiscale approach that relies on the method of multiple scales, which scales the forces such that they are similar in proportion.

The scaling approach has been used previously with two- and three- dimensional models, where the laser forces were computed by discretizing the laser beam into a

finite number of rays. However, this method of calculating laser forces starts to fall apart as the particle size becomes smaller than the wavelength of the laser. This work presents a three-dimensional model that relies on the Generalized Lorenz-Mie Theory (GLMT) to calculate laser forces.

An online constraint embedding method has been used in the past to enforce the normality constraint of the Euler parameters in the rotational dynamics of the 3D model. However, this method requires the integrator to be stopped in order to change the dependent Euler parameter. This work provides a novel elimination approach that does not require the definition of the dependent Euler parameter. Moreover, this approach does not require extra equations to be solved during the integration, which results in lower computation time.

The numerical solvers used to integrate the Equations of Motion (EOMs) change both the type of result and the computational requirement. A micro- or nano- particle observes forces due to the Brownian motion in the surrounding fluid. The inclusion of such forces makes the EOMs of the particle stochastic differential equations (SDEs) instead of the ordinary differential equations (ODEs). Moreover, the disproportionality in the inertia and fluid forces makes these equations stiff in nature. Thus, the stiff SDE solvers from the Julia programming language are used here to tackle both of these scenarios. The use of these solvers along with the scaling technique helps achieve further reduction in computation time.

The inclusion of Brownian motion force and the use of stochastic differential equation solvers allow the analysis of the problem in the frequency domain. It is common practice with optical trapping experiments to measure the trap stiffness with the use of Power Spectral Density (PSD) analysis. This work develops a theoretical equation of the PSD of the nanoparticle's position along with scaling. This helps estimate the PSD of the nanoparticle that would show the underdamped behavior in the experiments.

Finally, a completely new experimental setup is prepared to investigate the claim of regime change from previous experiments in further detail. The experimental setup is designed to achieve easy and repeated trapping of the nanoparticle along with high frame rate recording of its trajectory. Software for all the necessary hardware, i.e. the high-speed camera, laser shutter and the back illumination system, is developed in-house to automate performing the experiments.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

This dissertation presents new findings and developments in the field of optical trapping of micro/nano particles. The focus is on both the simulation models and the experiments. Such simulations may involve modeling a free-floating body that has six degrees of freedom. This raises the issue of treating the spatial rotations, which requires special treatment beyond the use of Euler Angles. Chapter 2 proposes a novel constraint elimination approach that handles the treatment of rotational dynamics by using unit quaternions (Euler parameters) instead of Euler angles. The complete simulation model for a particle in an optical trap is introduced in Chapter 3. It discusses (1) different methods of computing laser forces, (2) the use of model scaling technique to simulate small bodies at large time scales, (3) the use of stiff solvers to improve the computational time, and (4) the inclusion of the stochastic term in the differential equations and their solution. The scaling method used to speed up the simulations relies on a scaling factor. It was commonly chosen by matching the experimental and simulation data. Chapter 4 investigates the underdamped motion observed in a previous dataset. The Power Spectral Density analysis (PSD), which is regularly used to measure the optical trap stiffness, is also introduced in Chapter 4 as a tool to estimate the scaling factor without comparing the simulations with the experiments. The introduction of the PSD analysis in the simulation allowed the prediction of the PSD of the 500nm diameter particle in an optical trap that showed the suspected underdamped motion in old experimental data. However, this data lacked the time history and temporal resolution to perform PSD analysis. Chapter 5 discusses the new experimental setup developed to collect new data and tries to explore the underdamped motion previously observed, with greater detail.

1.2 Treatment of rotational dynamics

The spatial orientation of a rigid body is often modeled with the help of three independent parameters, e.g. Euler Angles. This approach works fine when the body's rotation is limited in range. For example, the dynamics of a quad-rotor, the dynamics of an aircraft, etc. However, there exist systems that require proper treatment of the attitude dynamics since their rotation is not limited to a range. Some common examples are free-floating micro and nano particles[46], spacecraft[55] or astronomical systems[2]. The use of Euler Angles with such systems creates a problem of 'Gimbal Lock' [38], where the ability to extract all three Euler angles from the rotation matrix of the object is lost. Other such sets of three independent parameters were also developed, e.g. classical or modified Rodrigues parameters[86], that improve the working range of the rigid body orientation but do not eliminate the problem of 'Gimbal Lock' completely.

The problem of 'Gimbal Lock' can be avoided by using a set of four parameters, known as quaternions, which was introduced by W. E. Hamilton. However, the implementation of quaternions in attitude dynamics also needs the enforcement of a normality constraint during the numerical integration of the equations of motion (EOMs). The equations of rotational motion of a rigid body can be computed without enforcing the normality constraint of the Euler parameters. This results in an eight-valued state vector. However, the body is not considered rigid anymore since the unconstrained Euler parameters may represent a non-orthonormal rotation matrix.

This problem is known as Gauss' mutation as discussed in [69, 81] and is generally not desired for rigid body simulations.

The set of quaternions that follow this algebraic constraint is also known as Euler parameters. Several approaches toward enforcing the normality constraint use Lagrange Multipliers [67, 68, 94, 104, 92, 93, 82, 84], Hamiltonian dynamics [85, 11, 12], coordinate reduction and constraint embedding techniques [21, 58, 34], overparametrization techniques [40] or Lie groups and Lie algebra [102, 89, 3, 23, 60]. One of the earliest implementations was proposed by Nikravesh et al. [68, 67] using Lagrange multipliers. However, Vadali [94] later showed that the EOMs can be reformulated such that the Lagrange multiplier is equal to zero. Sherif [84] provided different formulations of rotational equations of motion when using Euler parameters and showed that they can be transformed into each other. Extending Vadali's work, Shabana [82] showed that the forces related to the normality constraint are identically equal to zero. Udwadia [92] initially provided the equations of motion using the fundamental equation of constrained motion. He later used this approach to develop a rotational controller [93] as well. All the aforementioned approaches yield a full-rank mass matrix, but the additional equations and variables in the EOMs mean that they do not employ elimination.

Möller and Glocker in [58] propose a coordinate reduction approach. The quaternions (Euler parameters) were not normalized, which allows a violation of the rigid body constraint. Thus, instead of a normality constraint, a perfect bilateral constraint was implemented to enforce the rigid body constraint, which requires an extra variable per body. The resulting set of EOMs is expressed in differential algebraic equation (DAE) form, which is not minimal, although the equivalent mass matrix is full rank.

A more modern approach uses Lie algebra to integrate the equations of motion [102, 89, 3]. This is accomplished using a special Lie group integrator like Crouch-Grossman[23] or Runge-Kutta-Munthe-Kaas[60]. The integration step is performed on the special unitary group $SU(2)$ which inherently enforces the normality constraint without introducing singularities into the mass matrix.

The approach closest to the proposed one was given by Haghshenas-Jaryani et al. in [34]. It uses a constraint embedding technique [98] where the EOMs are reduced to a minimal set by choosing a dependent Euler parameter and eliminating it at the velocity and acceleration levels. However, this process requires a 'Switching Strategy' to avoid the singularity introduced in the mass matrix by this type of standard elimination.

In this work, a new elimination approach is proposed that takes advantage of Kane's Method's ability to redefine state space. This allows enforcement of the normality constraint at both the velocity and acceleration levels. It also reduces the computation time by maintaining the system's ODE form while eliminating the use of the 'Switching Strategy'.

1.3 Simulation of Micro/Nano particles in an optical trap

Many biomechanical systems are modeled as micro/nano scale rigid bodies moving in a fluid characterized by Stokes flow. An optical trap is one such system that is widely used today to hold and manipulate objects from a couple of micrometers[4] to a few nanometers[5] in size. Their ability to apply force in the piconewton scale and measure displacements at nanometer scale has made them useful in numerous applications, such as measuring the viscoelastic properties of cell membranes[31], study the physics of colloids[10], develop molecular motors[1], and measure mechanical properties of biopolymers[35] and microtubules[47].

The need for a simulation model for the particles trapped in an optical tweezer stems from the difficulty in measuring the particle's position in all three dimensions. Some modern techniques allow the measurement of the trapped particle in three dimensions[17, 71]. However, these rely on adding more equipment to and having good calibration of the experimental setup. A simulation model can help estimate the particle trajectory in the third dimension by matching the simulation results with the planar experimental data recorded using a camera.

Viscous drag force acting on a particle, Stokes' drag, is extremely large at the microscale and nanoscale compared to the body's own inertia force [72]. The large disproportion between the magnitudes of the inertia and viscous drag forces yields a stiff system. To simulate such systems easily, it is a common practice to ignore the inertia force, which results in a set of first-order ordinary differential equations (ODEs) [97] referred to as the overdamped Langevin equation [70]. This model can be solved fairly quickly using well-known Runge-Kutta methods [18]. However, it has been shown that modeling the inertia force is important because it can affect a small body's motion [33, 16, 13] over longer time periods.

Solving a stiff system is challenging since the integration time steps commonly end up being extremely small. The EOMs were previously[33] solved using an explicit solver. The problem of long computation time was tackled using a well-established scaling approach based on the Method of Multiple Scales (MMS) [61]. This approach takes advantage of the disproportionality in the drag and inertia forces to scale them such that they do not produce large accelerations, but achieve an accurate estimate of the system's motion at longer time periods. However, explicit solvers are not best suited for stiff systems. Implicit solvers, such as the trapezoid [32] or Rosenbrock [83] methods, are more suitable to such stiff problems and can solve them with much less computational cost. This work makes use of implicit solvers along with the scaling method described above to reduce the computation time even further.

Another force acting on the particle, that depends on the surrounding medium, is the random force associated with Brownian motion [54]. Albert Einstein first explained Brownian motion in 1905. Today, this motion is modeled as a random white-noise force. When the differential equation model contains a random variable, it is classified as a stochastic differential equation (SDE) model. Techniques used to solve ODEs cannot be directly applied to the SDEs because of the existence of a random variable [36]. The calculus required to solve SDEs was developed recently [41] as compared to that of ODEs[18]. The numerical methods to integrate SDEs are still an active area of research[78].

Previous simulations [33] treated the stochastic nature of the equations as a constant force over a short time period. This method generates trajectories similar to the correct solution. However, the method falls apart when the results are analyzed using frequency domain techniques. Moreover, it required the ODE solver to be stopped repeatedly which increased the computation time. Many numerical methods exist, which can adaptively solve stiff SDEs [39, 73]. This work uses the stiff SDE solvers available in `DifferentialEquations.jl` package of the Julia programming language to solve the EOMs of the particle.

Regardless of the choice of integration scheme, the model used for calculating the forces generated by the interaction between the particle and the laser can affect the final solution significantly. There exist three force models that have different accuracies based on the wavelength, λ, of the laser being used and the radius, r, of the particle. These three models are (1) Rayleigh scattering (when $r \ll \lambda$), (2) Ray Optics (when $r \gg \lambda$) and (3) Generalized Lorentz-Mie theory (when $r \approx \lambda$) [50].

The simulations were performed for particles with three different diameters, 500nm, 990nm and 1950nm. The laser wavelength used was 1064nm. Previous simulations [33] used the Ray Optics model to calculate laser force while the particle sizes were similar to that of the laser wavelength. The use of the Ray Optics model in such a case is not desirable since it overestimates the laser forces acting on the smaller particles. In this dissertation, Generalized Lorentz-Mie theory (GLMT) is used to calculate the laser forces more accurately.

The collective use of stiff stochastic differential equation solvers, the model scaling approach and the use of Generalized Lorentz-Mie theory for calculating the laser forces achieved significantly lower computation times, $\approx 21min$ in the previous study[33] for a single simulation to $\approx 1.4min$ in this work for 10000 simulations, along with a more realistic representation of the Brownian motion physics and the light-particle interaction.

1.4 Exploration of the underdamped motion

The scaling approach, discussed in the previous section, showed significant improvements in the computation time with the old model. However, it was important to validate the model with more experimental data. Few optical trapping experiments were performed with particles of diameter 500nm, 990nm and 1950nm. In these experiments, the particle's trajectory was recorded at a high frame rate while it was being trapped, i.e. as the particle was moving toward the focal point. These experimental trajectories are shown in Figs. 1.1, 1.2 and 1.3.

Conventional wisdom states that, at micro/nano scale, the viscous friction forces have a much larger impact on the particle's motion than inertial forces [72]. Thus, many works omit the mass properties from the model[22, 20, 14, 6, 48], particularly those that use the overdamped Langevin equations[91, 13]. As suggested by the

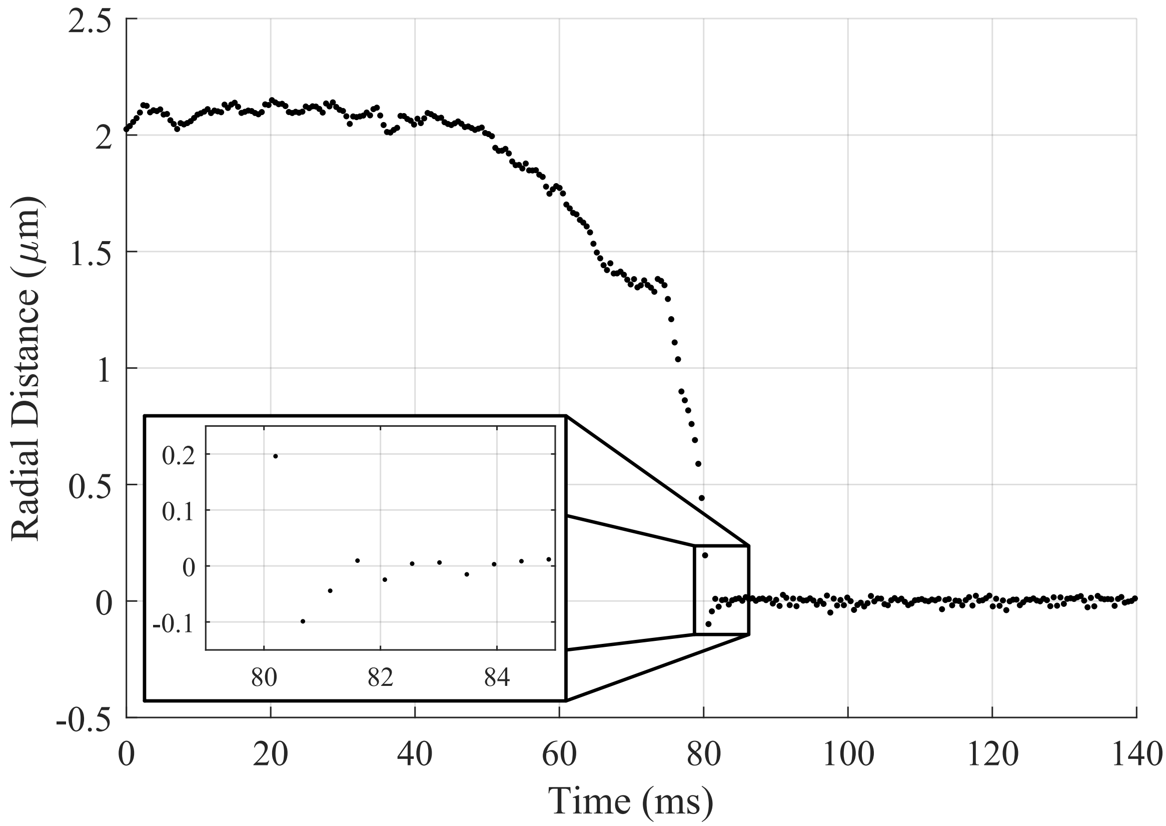


Figure 1.1: Trajectory of a 500nm diameter particle showing the underdamped behavior as being trapped

name, these equations predict overdamped motion as the particle approaches the focal point, i.e. the particle should not go beyond the focal point while moving towards it. However, in the experiments, the smaller particles exhibit what appears to be underdamped motion, i.e. particles go beyond the focal point and come back. This phenomenon seemed to get more pronounced as the particle size was reduced.

A hypothesis was made based on the experimental data, that there should be regime change in the dynamics of the particle as the size of the particle reduces below 1μm. However, only the simulation models with the use of the scaling approach captured this phenomenon. Thus, it was concluded that there must be a force model, either the laser, the fluid, or the inertia, that breaks down as the particle size reduces.

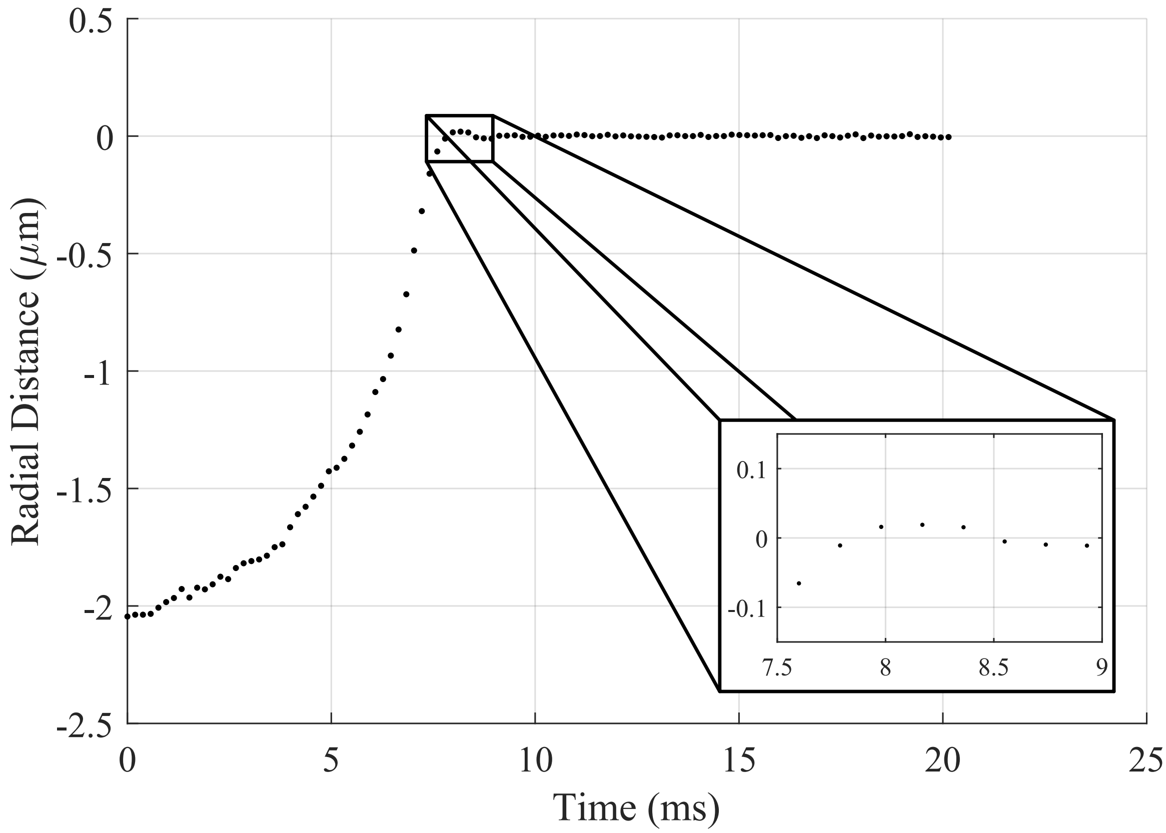


Figure 1.2: Trajectory of a 990nm diameter particle being trapped

The limiting factor in analyzing this phenomenon was the amount of experimental data that was available since only one dataset for each particle size was available.

A new experimental setup is developed here to conduct multiple optical trapping experiments in a short time period. It was also desirable to keep the experimental parameters close to the ones from the previous experiment. The final goal was to characterize the experimental conditions that would repeatedly achieve the suspected underdamped motion. Once such conditions are identified, a detailed analysis can be conducted on the reliable and repetitive data of the underdamped phenomenon to develop better force models.

The experimental setup developed here was able to trap polystyrene particles ranging from 2μm to 200nm in diameter. However, no underdamped behavior was

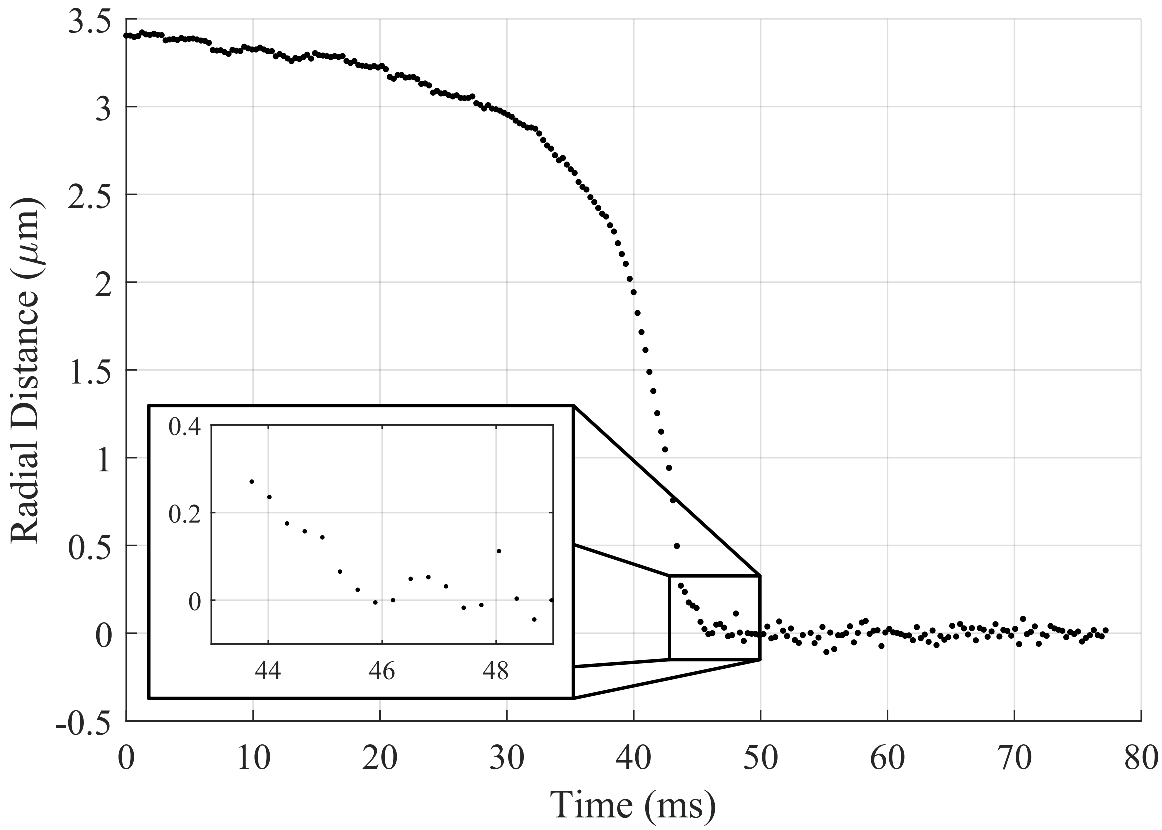


Figure 1.3: Trajectory of a 1950nm diameter particle showing the overdamped behavior as being trapped

observed in about 100 experiments that were conducted in total for different particle sizes. Thus, a decisive conclusion cannot be drawn at the moment regarding the underdamped motion since the result of the new experimental data is in contradiction with the old data.

CHAPTER 2

EULER PARAMETERS AND THE NORMALIZATION CONSTRAINT

2.1 Introduction

While implementing the Euler parameters in the equations of motion of a dynamic system, avoiding both the singularity in the rotation matrix and the use of differential-algebraic equations (DAEs) is a nontrivial task. Standard elimination can be used here which involves choosing a dependent variable and eliminating it from the EOMs at the velocity and acceleration levels. However, this method needs a 'Switching Strategy' to tackle the problem of choosing the dependent Euler parameter. This increases the computation time since the ODE solver needs to be restarted for the switch of the dependent Euler parameter to take place.

The approach proposed in this chapter uses Kane's method to develop the equations of motion. Here, the normality constraint is addressed using a novel elimination approach that does not require an explicit definition of one of the Euler parameters as a dependent variable. This process involves a redefinition of the state space in terms of an auxiliary speed, which is defined as the derivative of the normality constraint. Later on, the EOM associated with this *auxiliary speed* is omitted. This yields a minimal set of EOMs with a full-rank mass matrix and an inherent enforcement of the normality constraint at both the velocity and acceleration levels. Moreover, the energy is also conserved within numerical integration tolerances, which is a common way of checking the reliability of the solution. This is computationally faster compared to the Lagrange multiplier based approaches [68, 67, 94, 82] as the number of equations to be solved is lower.

2.2 Addressing the Normality Constraint

The discussion of the proposed approach is within the framework of Kane's method [45, 15] and relies on its definition of *auxiliary speeds*. Within this framework, *velocity* is considered to be a vector and *speeds* are its components. Auxiliary speeds are typically used to solve for non-contributing (reaction) forces, such as the Lagrange multipliers associated with the normality constraint. The EOMs associated with the auxiliary speeds can be omitted, yielding a set of relations equal in number to the number of independent generalized speeds (which are the time derivatives of generalized coordinates.)

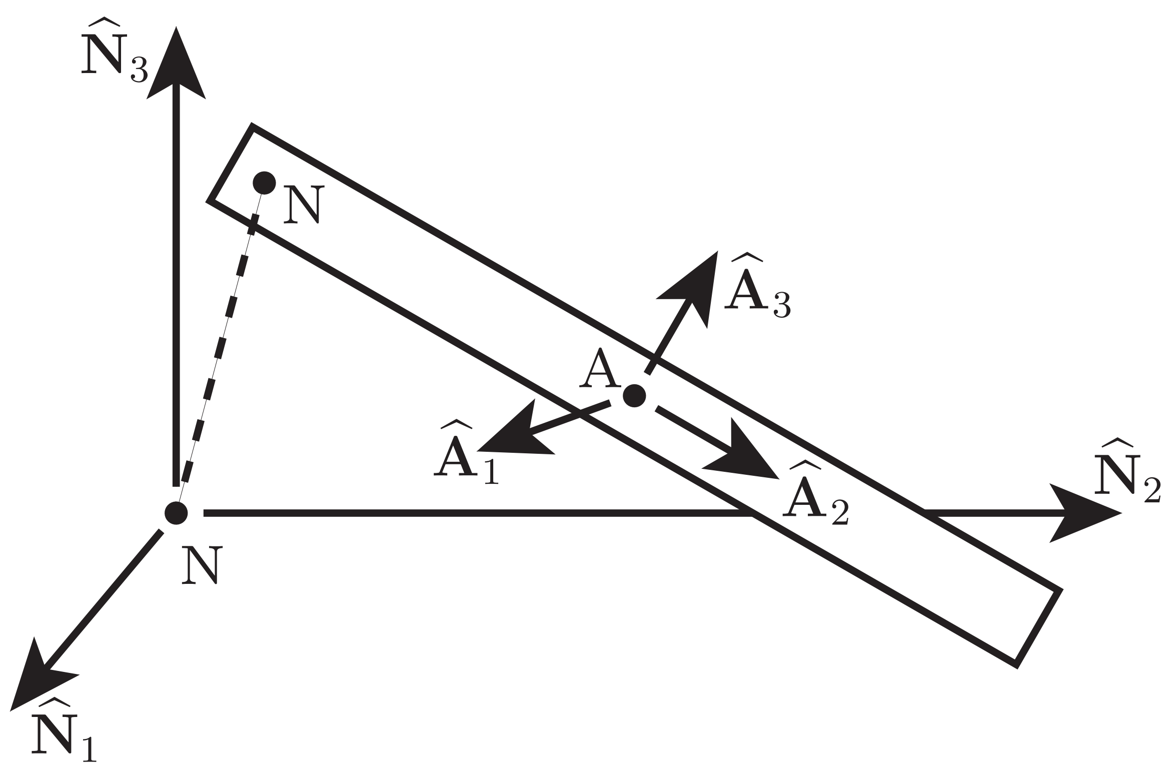


Figure 2.1: 3D single pendulum model

To illustrate this approach, consider the spatial pendulum with mass m_A in Fig. 2.1, with a body-attached frame A $= (\widehat{\mathbf{A}}_1, \widehat{\mathbf{A}}_2, \widehat{\mathbf{A}}_3)$ and a body-attached point A at its mass center. The pendulum has three rotational degrees of freedom (DOFs). The inertial reference frame is N = $(\widehat{\mathbf{N}}_1, \widehat{\mathbf{N}}_2, \widehat{\mathbf{N}}_3)$ and the inertial reference point is N. Body A is connected to the ground by a spherical joint at the inertial reference point N. If the vector of Euler parameters representing the spatial orientation of frame A observed from the inertial frame N is designated $\mathbf{e}$:

$$\mathbf{e} \;=\; \begin{bmatrix} e_0 & e_1 & e_2 & e_3 \end{bmatrix}^T \tag{2.1}$$

The angular velocity and the angular acceleration of this body observed from the inertial frame are well-defined in [25]:

$$^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \;=\; 2L(\mathbf{e})\,\dot{\mathbf{e}} \;=\; \dot{\theta}_1\;\widehat{\mathbf{A}}_1 \;+\; \dot{\theta}_2\;\widehat{\mathbf{A}}_2 \;+\; \dot{\theta}_2\;\widehat{\mathbf{A}}_3 \tag{2.2}$$

$$^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}} \;=\; 2L(\mathbf{e})\,\ddot{\mathbf{e}} \tag{2.3}$$

where $\dot{\mathbf{e}}$ and $\ddot{\mathbf{e}}$ are the time derivatives of the Euler parameters and

$$L(\mathbf{e}) \;=\; \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \end{bmatrix} \tag{2.4}$$

The position vector from the inertial point N to the mass center A and the velocity of the mass center can be given as

$$\mathbf{P}_{NA} \;=\; L_A\;\widehat{\mathbf{A}}_2 \tag{2.5}$$

$$\mathbf{V}_A \;=\; {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \mathbf{P}_{NA} \;=\; 2\,(\mathbf{P}_{NA}\otimes)^T\, L(\mathbf{e})\,\dot{\mathbf{e}} \tag{2.6}$$

where, L_A is the half of the length of the rod, and $\otimes$ is the cross product operator. For a 3×1 vector $\mathbf{X}$, the cross product operator is defined as

$$\mathbf{X}\otimes \;=\; \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\otimes \;=\; \begin{bmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{bmatrix} \tag{2.7}$$

Note that the matrix $\mathbf{X}\otimes$ is a singular matrix.

The normalization constraint that must be enforced has the form,

$$e_0^2 \;+\; e_1^2 \;+\; e_2^2 \;+\; e_3^2 \;=\; \mathbf{e}^T\mathbf{e} \;=\; 1 \tag{2.8}$$

A standard elimination approach can be used to address this problem, but that introduces a singularity. For example, if we choose $\dot{e}_0$ as the dependent speed, we can solve for it and eliminate it from subsequent equations:

$$\dot{e}_0 \;=\; -\frac{e_1\dot{e}_1 \;+\; e_2\dot{e}_2 \;+\; e_3\dot{e}_3}{e_0} \tag{2.9}$$

Equation (2.9) shows that eliminating $\dot{e}_0$ will introduce a singularity in the EOMs when $e_0 \to 0$, and similarly if any other Euler parameter is chosen. However, the Euler parameters must be constrained to prevent a singular mass matrix in the EOMs. Here it is shown that elimination can still be used with a careful definition of the state space.

To implement this constraint using the proposed approach, three *quasi-speeds* and one auxiliary speed can be defined as

$$\mathbf{u} \;=\; \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \;=\; \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \\ \dot{\theta}_4 \end{bmatrix} \;=\; \begin{bmatrix} -e_1 & e_0 & e_3 & -e_2 \\ -e_2 & -e_3 & e_0 & e_1 \\ -e_3 & e_2 & -e_1 & e_0 \\ e_0 & e_1 & e_2 & e_3 \end{bmatrix} \begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} \;=\; \begin{bmatrix} L(\mathbf{e}) \\ \mathbf{e}^T \end{bmatrix} \dot{\mathbf{e}} \;=\; E(\mathbf{e})\;\dot{\mathbf{e}} \tag{2.10}$$

where u_1, u_2 and u_3 are the quasi-speeds, and u_4 is the auxiliary speed; quasi-speeds are defined as the non-existence of an antiderivative. Note that $E(\mathbf{e})$ is an orthonormal matrix, therefore

$$\dot{\mathbf{e}} \;=\; E^T\;\mathbf{u} \tag{2.11}$$

The equations of motion for this system can be expressed in the following form:

$$\begin{bmatrix} F_1^* \\ F_2^* \\ F_3^* \\ F_4^* \end{bmatrix} = A(\mathbf{e})\,\dot{\mathbf{u}} + \mathbf{b}(\mathbf{u},\mathbf{e}) = \boldsymbol{\Gamma}(\mathbf{u},\mathbf{e}) = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \tag{2.12}$$

where $\mathbf{b}(\mathbf{u},\mathbf{e})$ is the vector of velocity forces, $A(\mathbf{e})$ is the mass matrix, which is rank deficient because of the use of Euler parameters (as discussed in the next paragraph), and $\boldsymbol{\Gamma}(\mathbf{u},\mathbf{e})$. The F_i are *generalized active forces* and the F_i^* are *generalized inertia forces.*

The mass matrix $A(\mathbf{e})$ for the system in Fig. 2.1 can also be calculated using following formula [15] for kinetic energy, T,

$$2T = \dot{\mathbf{e}}^T A(\mathbf{e})\,\dot{\mathbf{e}} = \dot{\mathbf{e}}^T \left[m_A\ J_{\mathbf{V}_A}^T\ J_{\mathbf{V}_A} + J_{{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}^T\ I_{\mathrm{AA}}\ J_{{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}\right]\dot{\mathbf{e}} \tag{2.13}$$

where, the jacobian matrices $J_{\mathbf{V}_A}$ and $J_{{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}$ are defined as

$$\begin{aligned} J_{\mathbf{V}_A} &= \begin{bmatrix} \frac{\partial \mathbf{V}_A}{\partial \dot{e}_0} & \frac{\partial \mathbf{V}_A}{\partial \dot{e}_1} & \frac{\partial \mathbf{V}_A}{\partial \dot{e}_2} & \frac{\partial \mathbf{V}_A}{\partial \dot{e}_3} \end{bmatrix} = 2\left(\mathbf{P}_{NA}\otimes\right)^T L(\mathbf{e}) \\ J_{{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}} &= \begin{bmatrix} \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}{\partial \dot{e}_0} & \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}{\partial \dot{e}_1} & \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}{\partial \dot{e}_2} & \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}}}{\partial \dot{e}_3} \end{bmatrix} = 2L(\mathbf{e}) \end{aligned}$$

Thus, the mass matrix can now be expressed as,

$$A(\mathbf{e}) = 4L(\mathbf{e})^T \left[m_A\left(\mathbf{P}_{NA}\otimes\right)\left(\mathbf{P}_{NA}\otimes\right)^T + I_{\mathrm{AA}}\right] L(\mathbf{e}) \tag{2.14}$$

Note that the matrix $L(\mathbf{e})$ is a 3×4 matrix with the maximum rank of 3. While, $\left[m_A\left(\mathbf{P}_{NA}\otimes\right)\left(\mathbf{P}_{NA}\otimes\right)^T + I_{\mathrm{AA}}\right]$ is a 3×3 matrix with rank 3. For any two matrices A and B, it can be said that

$$\operatorname{rank}(AB) \le \min\left(\operatorname{rank}(A), \operatorname{rank}(B)\right) \tag{2.15}$$

From (2.15), it can be concluded that the resultant mass matrix has to have rank 3 or less with size 4×4. Thus, the mass matrix has to be singular. Another simple explanation for the singularity of the mass matrix is that a rigid body has a maximum of 3 degrees of rotational freedom. Thus, if the mass matrix is 4×4 then it has to be singular since the size of the mass matrix has to be $n \times n$, where n is the degrees of freedom. This conclusion applies directly to the single spatial pendulum in Fig. 2.1 but can be generalized to any system described in terms of the Euler parameters.

The set of dependent relations in Eqn. (2.12) can be reduced to an independent set by eliminating u_4 and $\dot{u}_4$ from the EOMs, using the normalization constraint on the Euler parameters

$$0 \;=\; \mathbf{e}^T \, \dot{\mathbf{e}} \;=\; u_4 \;=\; \dot{\theta}_4 \quad \rightarrow \quad 0 \;=\; \mathbf{e}^T \, \delta\mathbf{e} \;=\; \delta\theta_4 \tag{2.16}$$

where, $\delta\theta_4$ is the virtual displacement associated with the auxiliary speed, u_4.

This elimination is accomplished by considering the virtual work done on the system:

$$\delta W \;=\; 0 \;=\; \sum_{i\,=\,1}^{4} \left(F_i - F_i^*\right)\; \delta\theta_i \tag{2.17}$$

where, $\delta\theta_i$ are the virtual displacements associated with the quasi-speeds, $u_{1:3}$, and the auxiliary speed, u_4. Combining Eqns. (2.16) and (2.17) yields,

$$0 \quad = \quad \left(F_1 - F_1^*\right)\delta\theta_1 + \left(F_2 - F_2^*\right)\delta\theta_2 + \left(F_3 - F_3^*\right)\delta\theta_3 + \left(F_4 - F_4^*\right)\cancelto{0}{\delta\theta_4} \tag{2.18}$$

which results in a reduced set of EOMs produced by eliminating u_4 and $\dot{u}_4$:

$$\begin{bmatrix} F_1^* \\ F_2^* \\ F_3^* \end{bmatrix} \;=\; \widetilde{A}\left(\mathbf{e}\right) \begin{bmatrix} \dot{u}_1 \\ \dot{u}_2 \\ \dot{u}_3 \end{bmatrix} + \widetilde{\mathbf{b}}\left(\mathbf{u},\mathbf{e}\right) \;=\; \widetilde{\mathbf{\Gamma}}\left(\mathbf{u},\mathbf{e}\right) \;=\; \begin{bmatrix} F_1 \\ F_2 \\ F_3 \end{bmatrix} \tag{2.19}$$

where $\widetilde{\mathbf{b}}\left(\mathbf{u},\mathbf{e}\right) \;=\; \mathbf{b}_{1:3}\left(\mathbf{u},\mathbf{e}\right)$ is the vector of velocity forces, $\widetilde{A}\left(\mathbf{e}\right) \;=\; A_{1:3\times1:3}\left(\mathbf{e}\right)$ is the mass matrix, and $\widetilde{\mathbf{\Gamma}}\left(\mathbf{u},\mathbf{e}\right) \;=\; \mathbf{\Gamma}_{1:3}\left(\mathbf{u},\mathbf{e}\right)$ is the vector of *generalized active forces.* This process of constraint elimination depends on a more general constraint embedding approach discussed in Appn. A.2.

The governing kinematic and dynamic equations of the system can thus be written in the following state space format

$$\begin{bmatrix} \dot{\mathbf{e}} \\ \\ \dot{\mathbf{u}}_{1:3} \end{bmatrix} = \begin{bmatrix} E^T \ \mathbf{u} \\ \\ \tilde{A}^{-1} \left(\tilde{\mathbf{\Gamma}} \ - \ \tilde{\mathbf{b}} \right) \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{e}} \\ \\ {}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}}/2 \end{bmatrix} \tag{2.20}$$

where, $\dot{u}_4 = u_4 = 0$. This process can be generalized to any number of sets of Euler parameters as long as the angular velocity considered reflects the frames that define the particular rotation defined by those Euler parameters. Also, note that the reduced $\dot{\mathbf{u}}$ vector in Eqn. (2.20) is equivalent to half of the angular acceleration in Eqn. (2.3) associated with the Euler parameters.

The transformation of the Euler speeds into quasi and auxiliary speeds yields a clear delineation between independent and dependent EOMs. Thus, the dependent EOM can be definitively eliminated; in contrast, it is unclear which Euler parameter to choose as dependent. The proposed process of eliminating u_4, $\dot{u}_4$ and $F_4 - F_4^* = 0$ (recall Eqns. (2.16) and (2.18)) is novel and yields a reduced set of EOMs with a full rank mass matrix. Thus, it is possible to eliminate the dependent Euler parameter without choosing one explicitly.

2.3 Simulation model

A three-dimensional double pendulum is used to illustrate the proposed method. The model has two bodies, two cylindrical rods with length L and radius r, as shown in Fig. 2.2. The two joints are spherical and so do not constrain spatial rotations of the links. The rod attached to the ground link has a body attached frame A while the rod with a free end has the body attached frame B. The inertial reference point

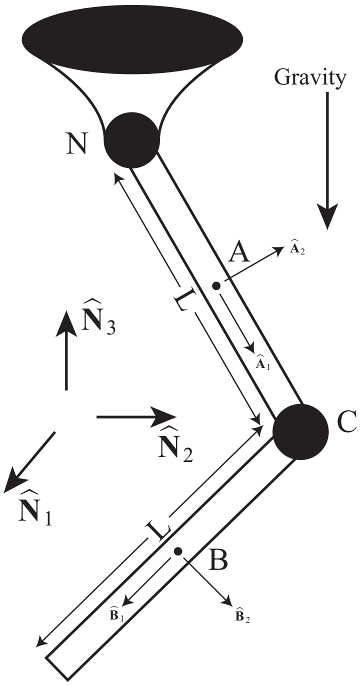


Figure 2.2: 3D double pendulum model

N is defined at the joint at the ground link and the inertial reference frame is N $= (\widehat{\mathbf{N}}_1, \widehat{\mathbf{N}}_2, \widehat{\mathbf{N}}_3)$. The mass centers of both bodies are points A and B respectively. Point C is defined at the center of the spherical joint between the two links. The vector of Euler speeds of body A with respect to frame N and of body B with respect to frame A and the corresponding quasi- and auxiliary speeds are defined as,

$$\dot{\mathbf{e}}_\mathrm{A} = \begin{bmatrix} \dot{e}_{A_0} & \dot{e}_{A_1} & \dot{e}_{A_2} & \dot{e}_{A_3} \end{bmatrix}^\mathrm{T} \qquad \mathbf{u}_\mathrm{A} = E\left(\mathbf{e}_\mathrm{A}\right)\dot{\mathbf{e}}_\mathrm{A}$$
$$\dot{\mathbf{e}}_\mathrm{B} = \begin{bmatrix} \dot{e}_{B_0} & \dot{e}_{B_1} & \dot{e}_{B_2} & \dot{e}_{B_3} \end{bmatrix}^\mathrm{T} \qquad \mathbf{u}_\mathrm{B} = E\left(\mathbf{e}_\mathrm{B}\right)\dot{\mathbf{e}}_\mathrm{B}$$

The equations of motion are formulated using the proposed approach, which is presented in detail in the Appn. A.1. The EOMs are also formulated using the method in [34] and the one where integration update is done on rotation matrices. The models are numerically integrated using the nonstiff variable step integrator `ode45` in MATLAB. This is a commonly used algorithm for numerical integration, which will

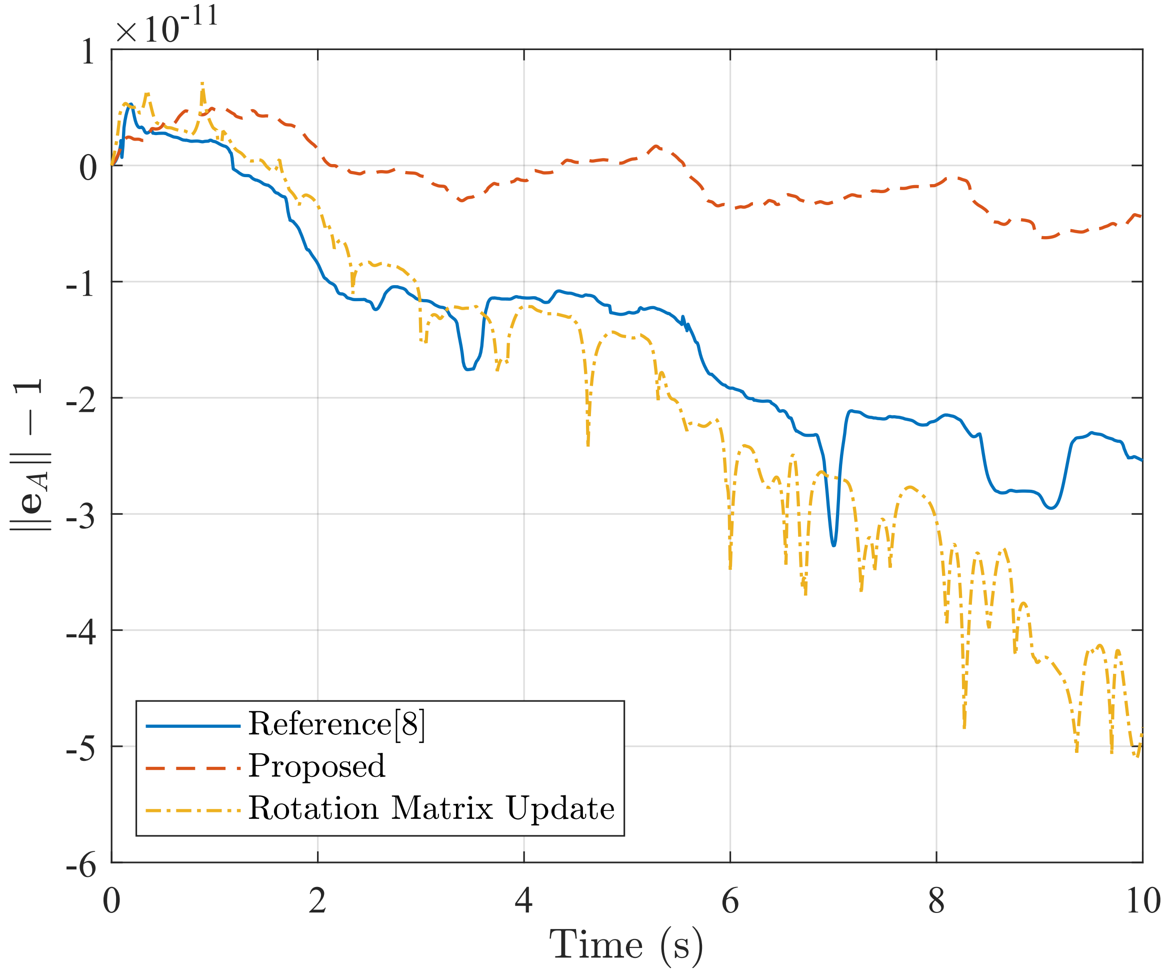


Figure 2.3: Normality of Euler parameters for body A (`RelTol` $= 1 \times 10^{-10}$, `AbsTol` $= 1 \times 10^{-9}$, $\Delta t = 0.01$)

be used here to examine the behavior of the proposed formulation of the EOMs. The system is integrated with 10^{-9} of absolute error and 10^{-10} of relative error tolerances. The system parameters and initial conditions are given in Table 2.1.

To validate the 3D double pendulum model, a check function based on the conservation of energy is implemented:

$$\Delta E \;=\; T_i + V_i - (T_0 + V_0) \tag{2.21}$$

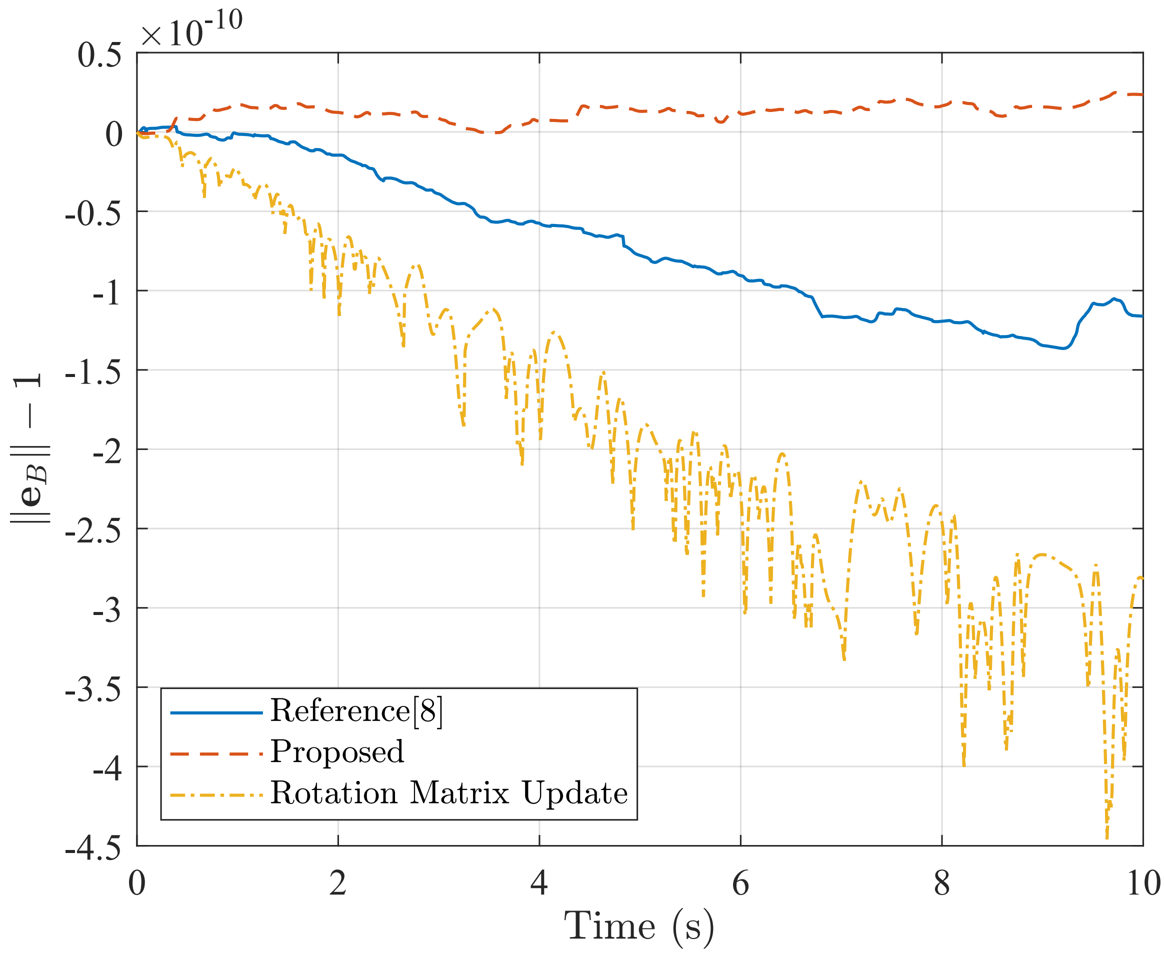


Figure 2.4: Normality of Euler parameters for body B ($\texttt{RelTol} = 1 \times 10^{-10}$, $\texttt{AbsTol} = 1 \times 10^{-9}$, $\Delta t = 0.01$)

where, T_0 and T_i are the kinetic energy at the initial condition and ith step of integration respectively, and V_0 and V_i are the potential energy at the initial condition and ith step of integration respectively. To conserve energy, the value of ΔE should remain close to zero, if there are no errors in system modeling. However, due to the computational errors, this value will not be exactly equal to zero. Thus, this function also serves as a tool to gauge the performance of the presented approach. The results of the simulation and the performance of the proposed approach are discussed in the next section.

Quantity	Value	Unit
L	0.3	m
r	0.05	m
m_A, m_B	1	kg
g	9.81	m/s^2
$\mathbf{e}_\mathrm{A}$	$\begin{bmatrix}0.27315297 & 0.18150666 & -0.15428524 & 0.93200796\end{bmatrix}^T$	NA
$\mathbf{e}_\mathrm{B}$	$\begin{bmatrix}-0.13787739 & 0.35886182 & 0.25501366 & 0.88722941\end{bmatrix}^T$	NA
$\dot{\mathbf{e}}_\mathrm{A}$	$\begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}^T$	NA
$\dot{\mathbf{e}}_\mathrm{B}$	$\begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}^T$	NA

Table 2.1: System parameters and initial conditions

2.4 Results and Discussion

This section provides the simulation results for the 3D double pendulum model discussed in the last section for the time history of 10s. The computational time for the proposed approach was $\approx$ 17 seconds. That for the approach in [34] was $\approx$ 56 seconds and for the rotation matrix update was $\approx$ 27 seconds. Note that the 3D double pendulum is a chaotic system. This means that a small error in computation can produce vastly different final system states. Thus, this system was chosen as a better performance measure among different techniques.

Herein, two metrics are used to analyze the accuracy of the proposed approach which are the normality satisfaction and the change in total energy. The normality satisfaction is checked by Eqn. (2.8) while the change in total energy is checked by Eqn. (2.21).

The values of the normality satisfaction are given in Figs. 2.3 and 2.4. Note that the norms for the proposed approach in Figs. 2.3 and 2.4 are in the order of 10^{-11} which is two magnitudes smaller than the absolute tolerance of 10^{-9}. However, that for the other approaches are in the order of 10^{-10}. The change in total energy is given in Fig. 2.5. For the proposed approach, it is on the order of the integration tolerance. While that for the other approaches is larger than the integration tolerance.

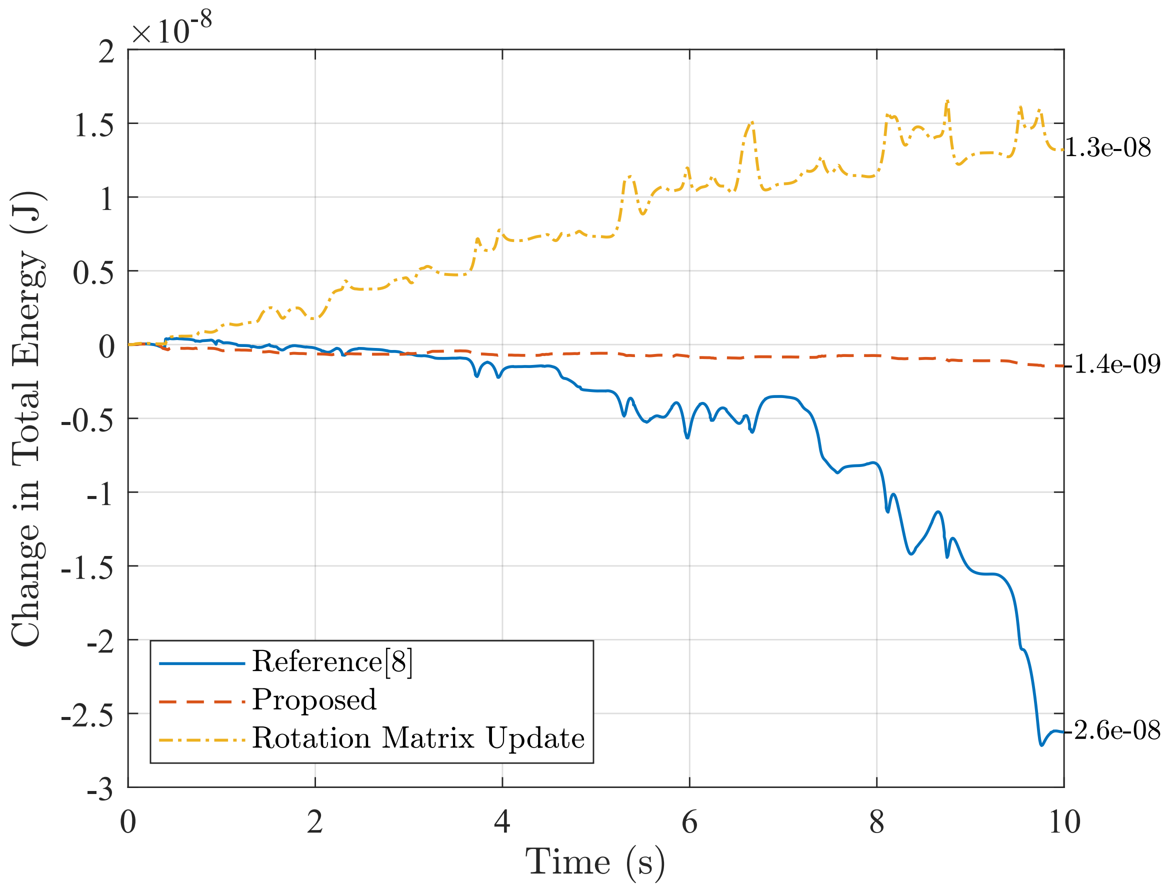


Figure 2.5: Change in total energy of the system for initial 10s ($\texttt{RelTol} = 1 \times 10^{-10}$, $\texttt{AbsTol} = 1 \times 10^{-9}$, $\Delta t = 0.01$)

A comparison of phase portraits, Euler parameters vs their derivatives, from all three approaches are provided in Figs. 2.6 and 2.7 for the time history of 10 seconds. Note that there is no observable difference in the system state trajectories and the final system states, among the three methods being compared. This also shows that the proposed approach does not affect the system behavior while providing results that are energetically more consistent. The values of Euler parameters at different times are provided in Tables 2.2, 2.3 and 2.4. Note that for all approaches, the Euler parameter values are the same for the first 8 seconds.

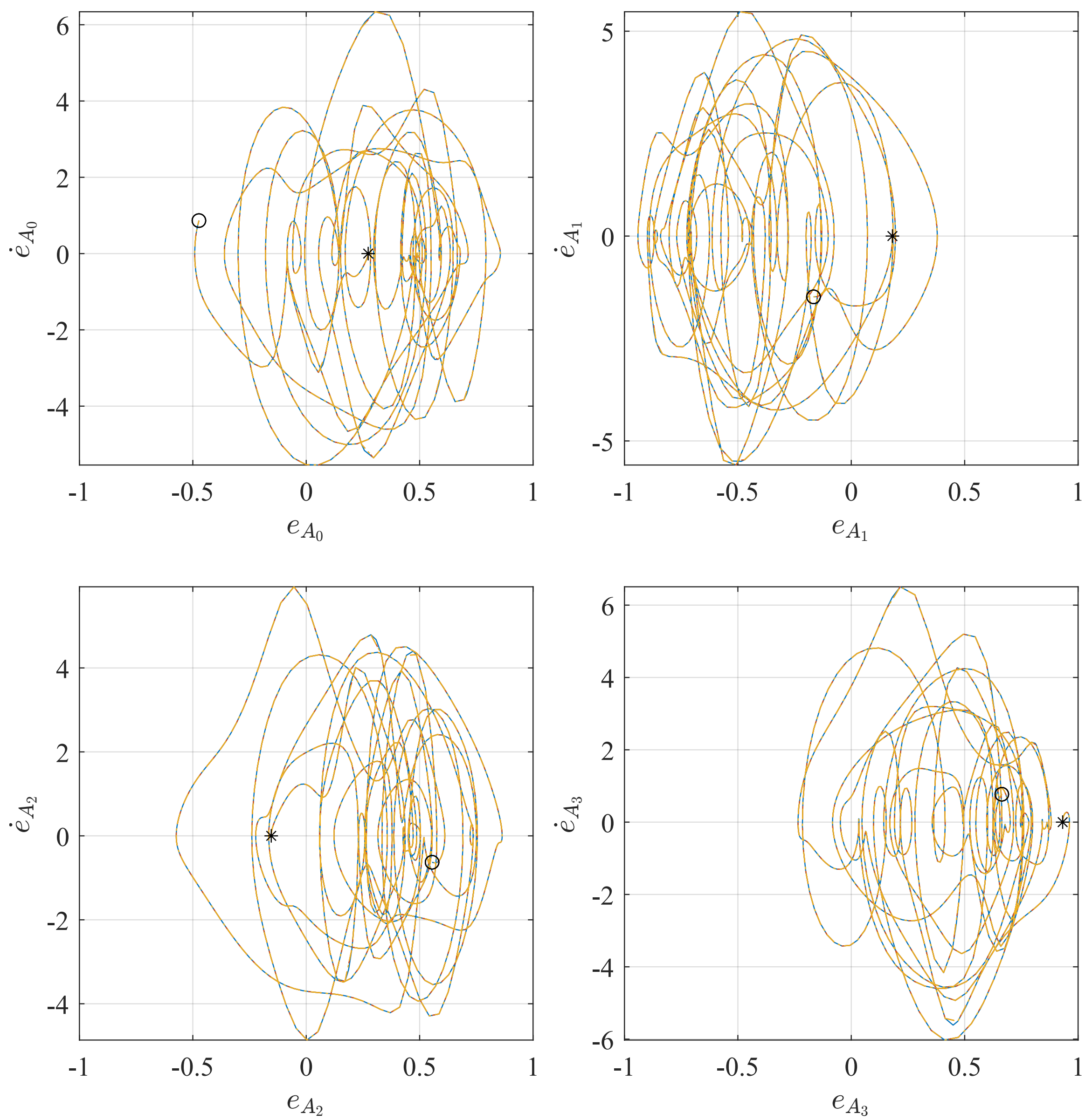


Figure 2.6: Phase portraits for body A Euler parameters (`RelTol` $= 1 \times 10^{-10}$, `AbsTol` $= 1 \times 10^{-9}$, $\Delta t = 0.01$, ——— Reference[34], - - - - Proposed, -·-·- Rotation Matrix Update, $*$ Initial State, $\circ$ Final State)

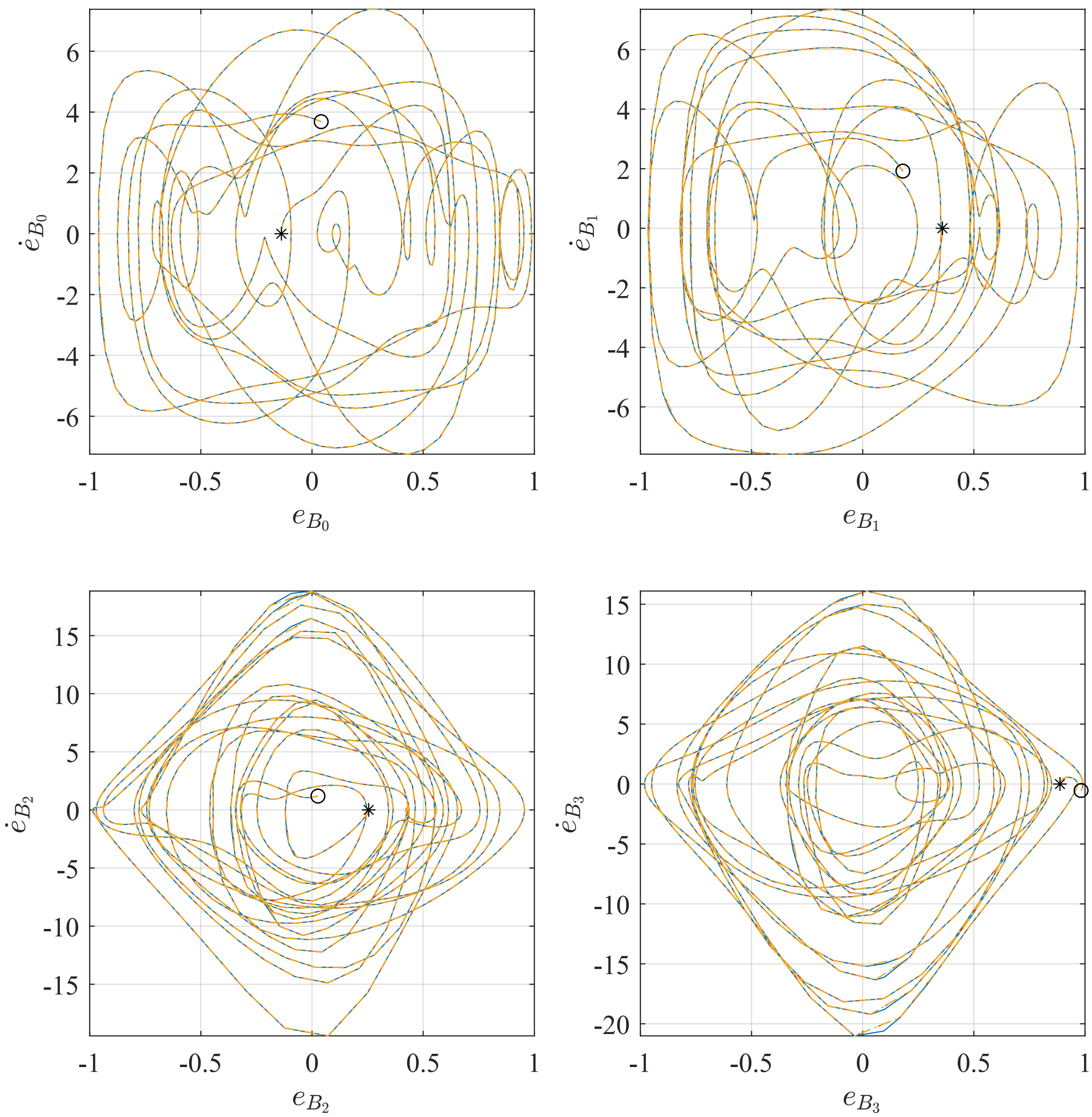


Figure 2.7: Phase portraits for body B Euler parameters (`RelTol` $= 1 \times 10^{-10}$, `AbsTol` $= 1 \times 10^{-9}$, $\Delta t = 0.01$, —— Reference[34], - - - - Proposed, -·-·- Rotation Matrix Update, $*$ Initial State, $\circ$ Final State)

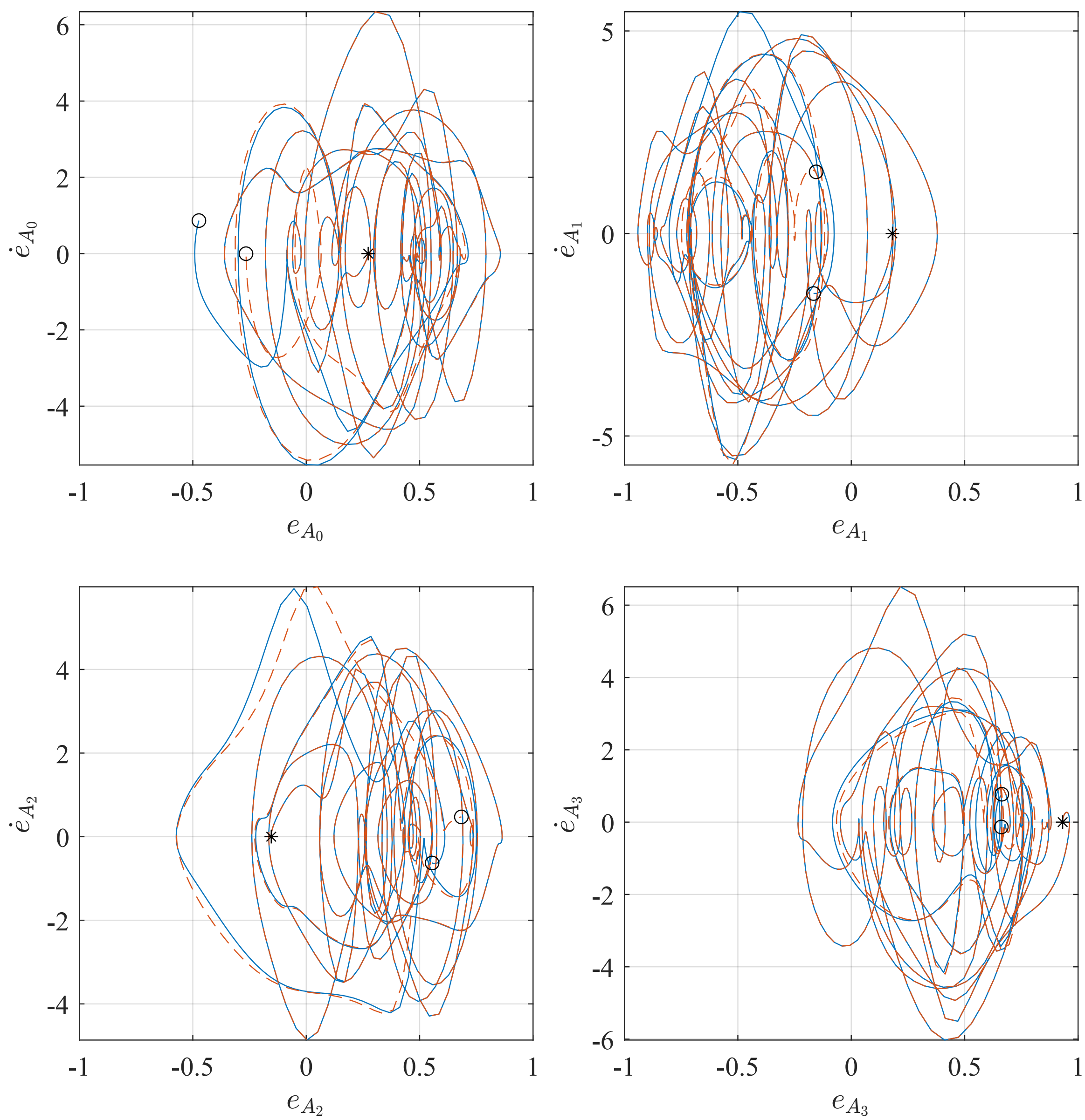


Figure 2.8: Phase portraits for body A Euler parameters (`RelTol` $= 1 \times 10^{-10}$, `AbsTol` $= 1 \times 10^{-9}$, $\Delta t = 0.01$, —— First Initial State (Table 2.1), - - - - Second Initial State (Eqn. (2.22)), $*$ Initial State, $\circ$ Final State)

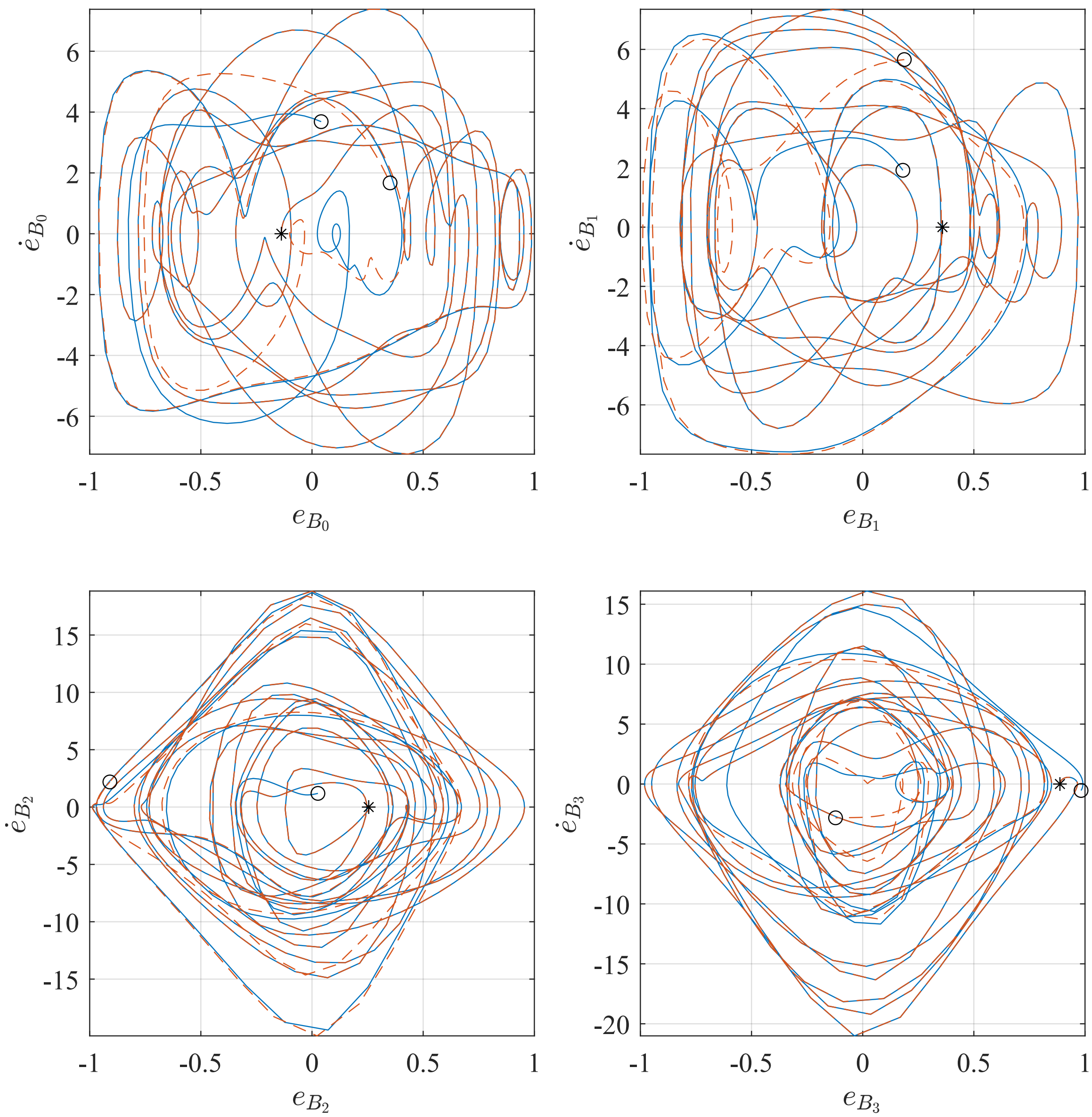


Figure 2.9: Phase portraits for body B Euler parameters (`RelTol` $= 1 \times 10^{-10}$, `AbsTol` $= 1 \times 10^{-9}$, $\Delta t = 0.01$, ——— First Initial State (Table 2.1), - - - - - Second Initial State (Eqn. (2.22)), $*$ Initial State, $\circ$ Final State)

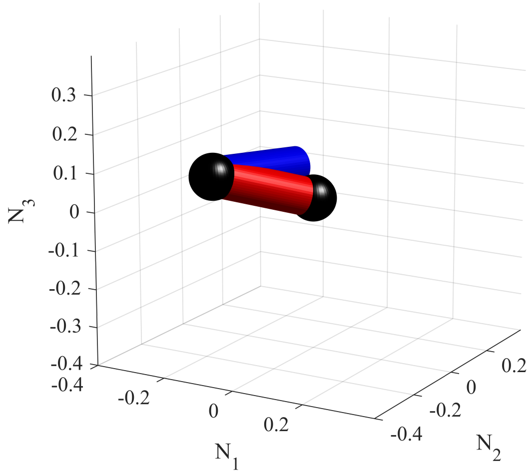


Figure 2.10: System Final state for initial condition in Table 2.1

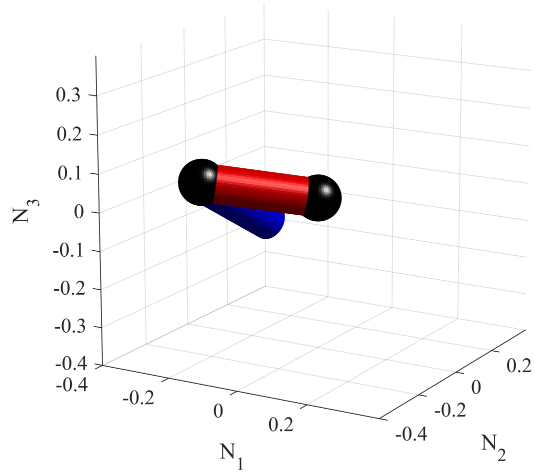


Figure 2.11: System Final state for the initial condition in Eqn. (2.22)

Another set of phase portraits is provided in Figs. 2.8 and 2.9. These phase portraits were generated using the proposed approach. However, the initial Euler parameter values differ slightly, which are given in Table 2.1 and Eqn. (2.22). Note that the final system states are very different for such a small change in the initial conditions of only one body. The final system states are shown in Figs 2.10 and 2.11. The characteristic nonlinear and chaotic behavior of the chosen system is observable in these phase portraits and is unchanged by the proposed approach.

$$
\begin{aligned}
\mathbf{e}_{\mathrm{A}} &= \begin{bmatrix} 0.2731520482 & 0.1815067144 & -0.1542852873 & 0.9320082195 \end{bmatrix}^T \\
\mathbf{e}_{\mathrm{B}} &= \begin{bmatrix} -0.1378773929 & 0.3588618275 & 0.2550136679 & 0.8872294192 \end{bmatrix}^T
\end{aligned}
\tag{2.22}
$$

2.5 Conclusion

This chapter showed that the proposed elimination approach is valid when using the Euler parameter representation of spatial orientation. The work showed that the normality constraint can be enforced without introducing singularities in the mass

Table 2.2: Euler parameters for approach in Ref. [34]

T(sec)	e_{A_0}	e_{A_1}	e_{A_2}	e_{A_3}	e_{B_0}	e_{B_1}	e_{B_2}	e_{B_3}
0	0.273153	0.181507	-0.154285	0.932008	-0.137877	0.358862	0.255014	0.887229
1	0.601657	-0.281299	0.249476	0.704728	-0.135954	0.961025	0.126427	-0.204852
2	0.172573	-0.830607	0.483631	-0.215432	-0.541611	-0.090728	0.584365	-0.597448
3	0.557720	-0.395463	0.480394	0.549344	-0.627860	0.733431	-0.043389	-0.256880
4	0.114685	-0.080346	0.471449	0.870705	0.699314	0.429925	-0.314040	-0.476973
5	0.490451	-0.854733	-0.163858	-0.045175	-0.371157	-0.378702	-0.697459	0.482055
6	0.513537	-0.523332	0.464964	0.496198	0.438356	-0.751934	-0.076442	0.486412
7	-0.355745	-0.102557	0.857771	0.356591	0.520118	-0.030659	0.542351	-0.659085
8	0.759543	-0.493050	-0.056060	0.420540	-0.381785	0.018261	-0.911075	0.154432
9	0.707391	-0.578222	-0.393869	0.100617	0.112072	-0.106242	-0.987751	-0.022389
10	-0.473064	-0.166345	0.554950	0.663755	0.041295	0.180773	0.026478	0.982301

Table 2.3: Euler parameters for proposed approach

T(sec)	e_{A_0}	e_{A_1}	e_{A_2}	e_{A_3}	e_{B_0}	e_{B_1}	e_{B_2}	e_{B_3}
0	0.273153	0.181507	-0.154285	0.932008	-0.137877	0.358862	0.255014	0.887229
1	0.601657	-0.281299	0.249476	0.704728	-0.135954	0.961025	0.126427	-0.204852
2	0.172573	-0.830607	0.483631	-0.215432	-0.541611	-0.090728	0.584365	-0.597448
3	0.557720	-0.395463	0.480394	0.549344	-0.627860	0.733431	-0.043389	-0.256880
4	0.114685	-0.080346	0.471449	0.870705	0.699314	0.429925	-0.314040	-0.476973
5	0.490451	-0.854733	-0.163858	-0.045175	-0.371157	-0.378702	-0.697459	0.482055
6	0.513537	-0.523332	0.464964	0.496198	0.438356	-0.751934	-0.076442	0.486412
7	-0.355745	-0.102557	0.857771	0.356591	0.520117	-0.030659	0.542351	-0.659085
8	0.759543	-0.493050	-0.056060	0.420540	-0.381786	0.018262	-0.911075	0.154429
9	0.707392	-0.578223	-0.393866	0.100615	0.112090	-0.106237	-0.987749	-0.022392
10	-0.473114	-0.166395	0.554901	0.663748	0.041267	0.180819	0.026667	0.982288

Table 2.4: Euler parameters when the rotation matrices are integrated directly

T(sec)	e_{A_0}	e_{A_1}	e_{A_2}	e_{A_3}	e_{B_0}	e_{B_1}	e_{B_2}	e_{B_3}
0	0.273153	0.181507	-0.154285	0.932008	-0.137877	0.358862	0.255014	0.887229
1	0.601657	-0.281299	0.249476	0.704728	-0.135954	0.961025	0.126427	-0.204852
2	0.172573	-0.830607	0.483631	-0.215432	-0.541611	-0.090728	0.584365	-0.597448
3	0.557720	-0.395463	0.480394	0.549344	-0.627860	0.733431	-0.043389	-0.256880
4	0.114685	-0.080346	0.471449	0.870705	0.699314	0.429925	-0.314040	-0.476973
5	0.490451	-0.854733	-0.163858	-0.045175	-0.371157	-0.378702	-0.697459	0.482055
6	0.513537	-0.523332	0.464964	0.496198	0.438356	-0.751934	-0.076442	0.486412
7	-0.355745	-0.102557	0.857771	0.356591	0.520117	-0.030659	0.542351	-0.659086
8	0.759543	-0.493050	-0.056061	0.420540	-0.381788	0.018262	-0.911075	0.154427
9	0.707393	-0.578225	-0.393864	0.100613	0.112105	-0.106232	-0.987748	-0.022395
10	-0.473157	-0.166439	0.554859	0.663741	0.041242	0.180859	0.026831	0.982277

matrix and without explicitly designating one of the Euler parameters as a dependent variable. A simulation showed that the proposed approach also yielded energetically consistent predictions because the changes in energy remained on or below the order of the numerical integration tolerance. Thus, the proposed approach provides a simple means of addressing the dependency in Euler parameters in a manner that results in an energetically consistent motion of the system.

CHAPTER 3

SIMULATION OF A PARTICLE IN AN OPTICAL TRAP

3.1 Introduction

This chapter presents the simulation model for a particle in an optical trap. Different forces acting on a particle are discussed here. This chapter also discusses different methods used for numerically integrating the equations of motion of the particle along with ways to reduce the computation time. The model scaling approach is also discussed here.

An optical trap is formed when a laser beam is focused with an objective lens of high numerical aperture (NA). A dielectric particle near the focal point experiences a force due to the momentum transfer and due to the electric field from the scattering of the laser. Figures 3.1(a),(b) and (c) describe the effect of a trap on a dielectric particle in terms of the total force due to a typical pair of rays R1 and R2 of the converging laser beam, under the assumption that the surface of the particle does not reflect light. In this approximation, the forces F1 and F2 are entirely due to the momentum change as a result of the refraction of incident rays R1 and R2. The forces are shown pointing in the direction of the momentum change.

Figure 3.1(a) and (b) showcase an arbitrary displacement of the particle origin P from the focal point f in the vertical direction, the magnitudes of F1 and F2 are the same and their vector sum gives a net restoring force F directed back to the focal point, and showcase the stability of the trap. Similarly, in Fig. 3.1(c), for arbitrary displacements of the particle center P from the focal point f in the horizontal direction, the magnitudes of F1 and F2 are different(shown by the change in thickness) and

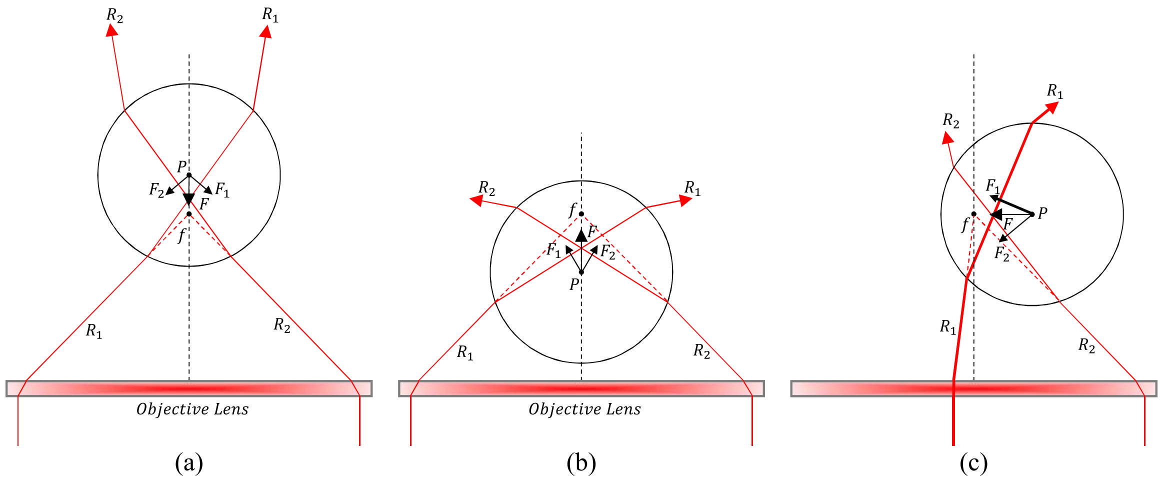


Figure 3.1: Arbitrary displacement of a trapped particle in the vertical direction, (a) and (b), and in the horizontal direction, (c)

their vector sum gives a net restoring force F directed back to the focal point. This difference in F1 and F2 arises from different intensities of the rays refracting through the particle. The shaded red bar at the bottom of Figs. 3.1(a),(b) and (c) represent the Laser's intensity profile which follows a Gaussian distribution.

3.2 Model Description and Equations of Motion

The simulation model is based on a simple dynamic model of a sphere in a fluid medium. The equations of motion for a particle trapped in an optical tweezer can be given as

$$A\ddot{\mathbf{q}} = \mathbf{\Gamma}\left(\mathbf{q}, \dot{\mathbf{q}}, t\right) \tag{3.1}$$

where, A is known as the mass matrix, $\ddot{\mathbf{q}}$ is a 3×1 generalized acceleration vector (also the cartesian acceleration in this case) and $\mathbf{\Gamma}\left(\mathbf{q}, \dot{\mathbf{q}}, t\right)$ is a 3×1 vector representing generalized forces. The generalized active forces for an optical tweezer can be expanded into

$$\mathbf{\Gamma}\left(\mathbf{q}, \dot{\mathbf{q}}, t\right) \;=\; \mathbf{F}_g \;+\; \mathbf{F}_b \;+\; \mathbf{F}_d \;+\; \mathbf{F}_l \;+\; \mathbf{F}_s \tag{3.2}$$

as shown in Fig 3.2. Where, $\mathbf{F}_g$ represents the gravitational force, $\mathbf{F}_b$ is the buoyant

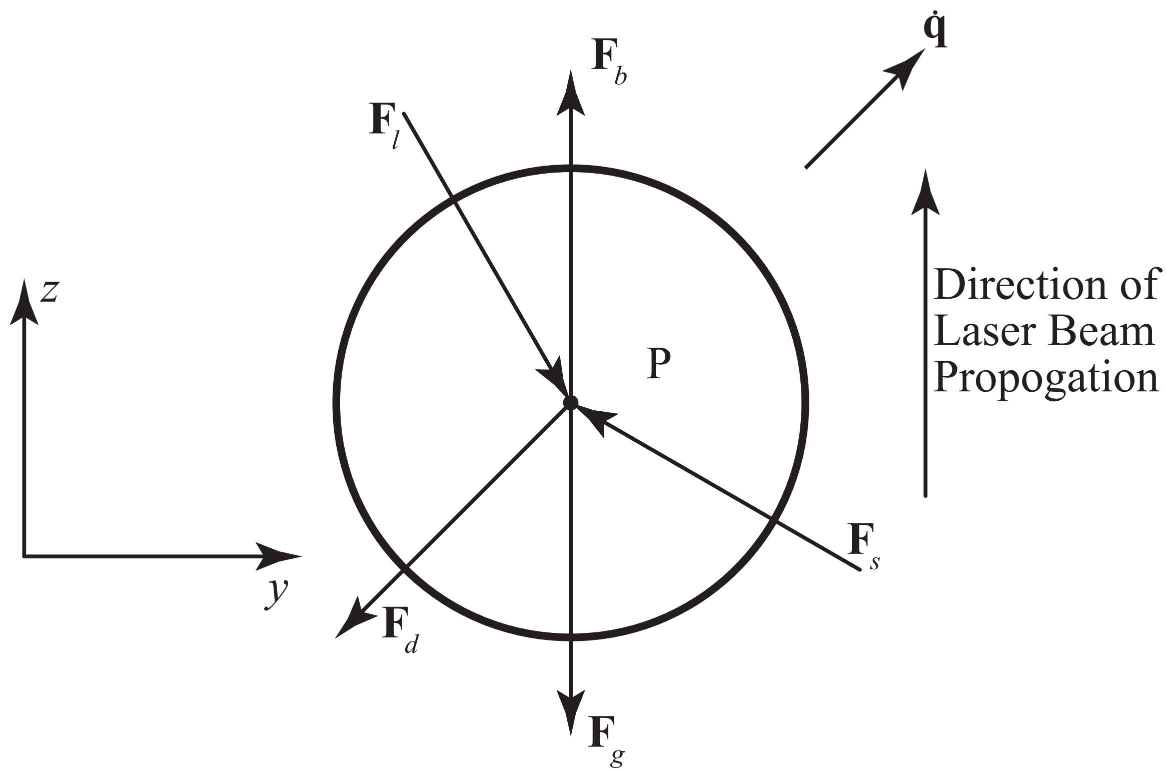


Figure 3.2: Forces acting on a particle in an optical trap

force, $\mathbf{F}_d$ represents the viscous drag on the particle by the fluid, $\mathbf{F}_l$ represents the effect of the laser beam interacting with the particle and $\mathbf{F}_s$ represents the random forces associated with Brownian motion.

3.2.1 Gravity and Buoyancy

The gravity and buoyancy forces are set as

$$\mathbf{F}_g = m\mathbf{g} = \begin{bmatrix} 0 & 0 & -mg \end{bmatrix}^T \qquad \mathbf{F}_b = -\rho_m V \mathbf{g} = \begin{bmatrix} 0 & 0 & \rho_m V g \end{bmatrix}^T \tag{3.3}$$

where, g is gravitational acceleration, ρ_m is the density of the surrounding medium and V is the volume of the surrounding medium displaced by the submerged particle.

It is clear from Eqn. (3.3) that the gravitational force act in the negative z-direction while the force due to buoyancy act in the positive z-direction.

3.2.2 Viscous Damping

To calculate the drag force acting on the particle, it is important to consider the characteristics of the surrounding medium. Knudsen number,Kn, can be used here to determine whether the fluid should be modeled as a continuum or as a discrete molecule system. The Knudsen number is defined as the ratio of the mean free path of the molecule of the surrounding medium, water here, and the characteristic length of the system under observation. The system under consideration has three different particle diameters 500nm, 990nm and 1950nm. The mean free path of the molecule of the surrounding medium is the same, $\lambda_{mfp} = 0.3$nm, for all three cases. This yields the Knudsen number of 0.0006 for the 500nm particle case. Since $Kn = 0.0006$, is less than 0.001, the surrounding medium can be considered as a continuum. Stokes' Law can be used to compute viscous drag force

$$\mathbf{F}_d = -\beta\dot{\mathbf{q}} = -6\pi\mu_m r \begin{bmatrix} \dot{q}_x & \dot{q}_y & \dot{q}_z \end{bmatrix}^T \tag{3.4}$$

where, $\dot{q}_x$, $\dot{q}_y$ and $\dot{q}_z$ are the translational velocities of the particle in each direction, μ_m is the dynamic viscosity of the fluid medium and r is the radius of the particle. It should be noted that the viscous drag acts opposite to the velocity of the particle.

3.2.3 Laser Beam Force

Beam force calculation for optical trap varies based on the size of the particle, r, relative to the wavelength, λ, of light used to trap it. At the macroscopic scale, $r \gg \lambda$, calculations based on the Ray-optics regime[4] are used. On the contrary, at the microscopic scale, $r \ll \lambda$, calculations based on the Rayleigh regime[88] are used. Calculating the beam force at the mesoscopic scale, $r \approx \lambda$, is mathematically the most challenging [50]. At this scale, the beam force can be calculated using the Generalized Lorentz-Mie theory (GLMT)[30, 65].

The total force from the laser beam is generally divided into two components. First is the scattering force, $\mathbf{F}_{l_s}$, which acts in the direction of the laser propagation. The second is the gradient force, $\mathbf{F}_{l_g}$, which acts in the direction of the laser intensity gradient. The most commonly used laser intensity profile is a Gaussian profile, also known as the lowest order Transverse Electromagnetic (TEM_{00}) mode[101], which is defined as[96],

$$I_0(\rho) = \frac{2P_t}{\pi w^2} \exp\left(-\frac{2\rho^2}{w^2}\right) \tag{3.5}$$

where, w is the beam radius, P_t is the total beam power and ρ is the radial distance from the center of the beam.

In the Rayleigh regime, since the particle is extremely small compared to the wavelength of the laser, it is modeled as a dipole in the electromagnetic field of the laser beam. The scattering force from the laser is proportional to the light intensity at the location of the particle while the gradient force is proportional to the gradient of the light intensity around the particle.

$$\begin{aligned} \mathbf{F}_{l_s} &= \frac{I_0 \sigma n_m}{c} & \sigma &= \frac{128\pi^5 r^6}{3\lambda^4}\left(\frac{m^2-1}{m^2+1}\right)^2 \\ \mathbf{F}_{l_g} &= \frac{2\pi\alpha}{c n_m^2}\nabla I_0 & \alpha &= n_m^2 r^3 \left(\frac{m^2-1}{m^2+2}\right) \end{aligned} \tag{3.6}$$

where, σ is known as the scattering cross section, α is known as the polarizability of the particle, c is the speed of light in vacuum, n_m is the refractive index for the medium, and m is the ratio of the refractive index for the particle (n_p) to the refractive index for the medium.

In the Ray-optics regime, the laser beam is discretized into several rays, and the approach of geometric optics is used to calculate the forces exerted by each ray on the particle. The contribution of each ray to the total scattering and gradient force on the particle is[4]

$$\begin{aligned}\mathbf{F}_{l_s} &= \frac{n_m P}{c}\left[1 + R\cos 2\theta - \frac{T^2\left[\cos(2\theta - 2\phi) + R\cos 2\theta\right]}{1 + R^2 + 2R\cos 2\phi}\right] \\ \mathbf{F}_{l_g} &= \frac{n_m P}{c}\left[R\sin 2\theta - \frac{T^2\left[\sin(2\theta - 2\phi) + R\sin 2\theta\right]}{1 + R^2 + 2R\cos 2\phi}\right]\end{aligned} \tag{3.7}$$

where, P is the power of the ray under consideration, θ and ϕ are the angle of incidence and the angle of refraction respectively, and R and T are the Fresnel Refraction and the Fresnel Transmission coefficients.

The third and more generalized approach known as the Generalized Lorenz-Mie theory (GLMT)[30, 65] computes the laser force on the particle using the conservation law of electromagnetic wave momentum. The total beam force, including both scattering and gradient, is computed by evaluating the flux of the Maxwell stress tensor through any virtual surface enclosing the object. The equation for this force is given as,

$$\mathbf{F}_l = \oint_S T_{ij} \cdot n_j dS \tag{3.8}$$

where, n_j is the unit vector normal to the enclosing surface, S, and T_{ij} is the Maxwell stress tensor. With recent advances in this field, it is easy to calculate the beam forces for both smaller and larger diameter particles[100]. However, it also comes with large computational costs exponentially increasing with the difference in particle size and wavelength.

The simulations and experiments provided in this work are done on particles with diameters of 500nm, 990nm and 1950nm. The wavelength of the laser used is 1064nm. Note that all three particle diameters are very close to the wavelength of the laser. Thus, this work uses a MATLAB toolbox[64, 53] for computing the laser beam forces using GLMT.

3.2.4 Brownian Motion Force

Stochastic forces are modeled to represent the thermal noise and interactions between modeled and non-modeled bodies. These forces essentially represent the Brownian motion. It is modeled as random forces acting at the mass center of the particle. They are implemented as Gaussian White noise. The random forces are defined as

$$\mathbf{F}_s = \sqrt{2k_BT\beta}\boldsymbol{\eta}(t) = \sqrt{2k_BT\beta}\begin{bmatrix}\eta_x(t) & \eta_y(t) & \eta_z(t)\end{bmatrix}^T \tag{3.9}$$

where, each component of random force, $\eta_i(t)$, is an independent random Gaussian process where for all t and t'[80]

$$E\left[\eta_i(t)\right] = 0 \qquad\qquad E\left[\eta_i(t)\eta_i(t')\right] = \delta(t-t') \tag{3.10}$$

The above equations imply that the mean value, $E\left[\eta_i(t)\right]$, of the forces associated with the Brownian motion is zero and the values at two distinct times, t and t', don't correlate, $E\left[\eta_i(t)\eta_i(t')\right]$, with each other.

3.2.5 Complete Model

Substituting the equations for all the forces into the equation of motion for the particle (3.1) yields

$$m\ddot{\mathbf{q}} = \underbrace{m\mathbf{g}}_{\mathbf{F}_g} - \underbrace{\rho_m V\mathbf{g}}_{\mathbf{F}_b} - \underbrace{6\pi\mu_m r\dot{\mathbf{q}}}_{\mathbf{F}_d=\beta\dot{\mathbf{q}}} - \underbrace{K\mathbf{q}}_{\mathbf{F}_l} + \underbrace{\sqrt{2k_BT\beta}\boldsymbol{\eta}(t)}_{\mathbf{F}_s} \tag{3.11}$$

Note that the laser force is added here as a linear spring using a stiffness matrix K. This is true when the particle is in a close vicinity of the focal point[62], regardless of the approach used for calculating the laser beam force. This approximation helps with some simple analysis performed in later sections. However, the simulations are still performed by calculating the laser force using GLMT.

3.3 Model Scaling Approach

A quick analysis of Eqn. (3.11) shows that a 2μm diameter glass particle will have the mass of $m = 8.3776 \times 10^{-15}\ kg$ and the damping coefficient of $\beta = 1.8887 \times 10^{-8}\ \frac{Ns}{m}$. This imbalance between mass and drag coefficient, $\mathcal{O}(10^{-7})$, makes the system stiff. The proposed method tackles the stiffness problem by first determining a small number from the model, which in this case is $m/\beta = 4.4356 \times 10^{-7}\ s$. Rewriting Eqn. (3.11) yields,

$$\mathbf{0} = \varepsilon\,(1\ s)\,\ddot{\mathbf{q}} - \frac{1}{\beta}\mathbf{F} + \dot{\mathbf{q}} = \varepsilon\ddot{\bar{\mathbf{q}}} - \frac{1}{\beta}\mathbf{F} + \dot{\mathbf{q}} \tag{3.12}$$

where $\varepsilon = 4.4356 \times 10^{-7}$ is unitless and $\mathbf{F}$ is a sum of all the forces except the drag force. This small parameter ε is used to introduce the slower time scales as

$$T_0 = \varepsilon^0 t \qquad T_1 = \varepsilon^1 t \qquad T_2 = \varepsilon^2 t \qquad \ldots \qquad T_n = \varepsilon^n t \tag{3.13}$$

The time derivatives $\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ can now be expanded into an asymptotic series as

$$\dot{\mathbf{q}} = \frac{d\mathbf{q}}{dt} = \varepsilon^0\frac{\partial \mathbf{q}}{\partial T_0} + \varepsilon^1\frac{\partial \mathbf{q}}{\partial T_1} + \varepsilon^2\frac{\partial \mathbf{q}}{\partial T_2} + \ldots \tag{3.14}$$

and

$$\ddot{\mathbf{q}} = \frac{d^2\mathbf{q}}{dt^2} = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty}\varepsilon^i\varepsilon^j\frac{\partial^2 \mathbf{q}}{\partial T_i \cdot \partial T_j} \tag{3.15}$$

Substituting these expansions into Eqn. (3.12) and rearranging the terms in the increasing order of ε yields

$$\mathbf{0} = \varepsilon^0\left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}\,(\mathbf{q}, t)}{\beta}\right) + \varepsilon^1\left(\frac{\partial^2 \bar{\mathbf{q}}}{\partial T_0^2} + \frac{\partial \mathbf{q}}{\partial T_1}\right) + \ldots \tag{3.16}$$

Note that the first term on the right side of Eqn. (3.16) contains all generalized active forces. The second term contains generalized inertia forces and the first set of higher-order terms. Considering how ε is defined, the difference between $\varepsilon^0 = 1$ and $\varepsilon^1 = 4.4356 \times 10^{-7}$ is fairly large, and so, it is necessary for the first term to largely cancel if the right side of Eqn. (3.16) is equal to zero. From the standpoint of multi-body dynamics, if forces cancel each other, they do no work and produce no motion. Such forces can thus be omitted from equations of motion.

The scaling of the generalized active forces is achieved by decomposing the first term of Eqn. (3.16) into small and large parts as

$$\varepsilon^0 \left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}(\mathbf{q}, t)}{\beta} \right) = (a_1 + a_2) \left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}(\mathbf{q}, t)}{\beta} \right) \tag{3.17}$$

where, $a_1 + a_2 = \varepsilon^0 = 1$ and $a_1 \gg a_2$. Substituting Eqn. (3.17) back into Eqn. (3.16) yields

$$\mathbf{0} = a_1 \left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}(\mathbf{q}, t)}{\beta} \right) + a_2 \left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}(\mathbf{q}, t)}{\beta} \right) + \varepsilon^1 \left(m \frac{\partial^2 \bar{\mathbf{q}}}{\partial T_0^2} + \frac{\partial \mathbf{q}}{\partial T_1} \right) + \dots \tag{3.18}$$

As discussed, the assumption here is that the large part of generalized active forces $\mathbf{\Gamma}(\mathbf{q}, \dot{\mathbf{q}}, t)$ cancel to the extent that it can be removed from Eqn. (3.18), yielding a second-order model of the form

$$\mathbf{0} = a_2 \left(\frac{\partial \mathbf{q}}{\partial T_0} - \frac{\mathbf{F}(\mathbf{q}, t)}{\beta} \right) + \varepsilon^1 \left(\frac{\partial^2 \bar{\mathbf{q}}}{\partial T_0^2} + \frac{\partial \mathbf{q}}{\partial T_1} \right) + \dots \tag{3.19}$$

$$\therefore \mathbf{0} = \beta \varepsilon \ddot{\mathbf{q}} - a_2 \mathbf{F}(\mathbf{q}, t) + a_2 \beta \dot{\mathbf{q}} \tag{3.20}$$

$$\therefore m(\mathbf{q}) \ddot{\mathbf{q}} + a_2 \beta \dot{\mathbf{q}} = a_2 \mathbf{F}(\mathbf{q}, t) \tag{3.21}$$

Note that the differential equation above is similar to Eqn. (3.12) except here the external forces are scaled by a scalar, a_2.

Quantity	Definition	Units	Value
T	Temperature	K	293.15
k_B	Boltzmann Constant	$\text{kg} \cdot \text{m}^2/\text{s}^2 \cdot \text{K}$	1.38×10^{-23}
ρ_s	Particle density	kg/m^3	2000
r_s	Particle radius	m	1.00×10^{-6}
g	Gravitational Acceleration	m/s^2	9.80665
ρ_m	Density of fluid	kg/m^3	998.2071
μ_m	Dynamic viscosity of fluid	$\text{kg}/\text{m} \cdot \text{s}$	0.001002
n_s	Refractive index of particle	Unitless	1.45
n_m	Refractive index of fluid	Unitless	1.33
c	Speed of light in a vacuum	m/s	299792458
NA	Numerical aperture	Unitless	1.2
λ	Laser wavelength	m	1.06×10^{-6}
P_0	Laser Beam Power	$\text{kg} \cdot \text{m}^2/\text{s}^2$	3.00×10^{-1}
$\mathbf{q}$	Initial position	m	$[-0.16\ 0.35\ 0.0] \times 10^{-6}$
$\dot{\mathbf{q}}$	Initial velocity	m/s	$[0.0\ 0.0\ 0.0] \times 10^{-6}$

Table 3.1: System parameters and initial conditions used for the simulation

3.4 Simulation and Experimental Setup

The scaled model for a particle in an optical tweezer, developed in the previous section, should not be integrated using an ODE solver. This is because it has a stochastic term in the form of the Brownian Motion force, $\mathbf{F}_s$. Here, the scaled system of differential equations, Eqn. (3.21), is solved using five different SDE solvers available in Julia programming language's package `DifferentialEquation.jl`. Moreover, the statistical nature of the SDEs implies that the solution of these SDEs will have a mean and variance. There are two possible approaches to achieve such a solution. One is by running numerous simulations and then averaging the results to obtain the mean and the variance[49]. An alternative approach to this is to use the Fokker-Planck equation[74] which operates in the probability space. The Fokker-Planck equation is a partial differential equation (PDE) that, when solved, results in the probability density function of the particle's position over time. The solution may require the use of analytical or numerical approaches[103] to solving PDEs depending on the external forces acting on the system. Here, the first approach is used. To achieve good average and variance values, each solver computes 10000 trajectories with the same initial condition for both scaled and unscaled systems. The initial condition and the system parameters are given in Table 3.1. The simulations were run on an AMD Ryzen 5 5600X CPU with 32GB RAM. A sample of the code to run sich simulations is available in Appn. F.

The experimental data was obtained by optical trapping of a 2μm diameter silica particle. The experiments were conducted by Dr. Peter Goodwin at the Center for Integrated Nanotechnologies (CINT) at the Los Alamos National Laboratory (LANL). A simplified version of the experimental setup used at CINT to obtain the experimental data is shown in Fig. 3.3. The detection laser, operating at 670nm, is used to illuminate the particle and capture its position using a Quadrant Photodiode Detector (QPD). Filtered green light from an incandescent lamp was used to illuminate the field of view of a camera that simultaneously recorded the trajectory of the particle falling into the optical trap. It should be noted that these devices can only accurately measure the particle's position in the focal plane, which is perpendicular to the direction of laser propagation at the sample stage. After a particle is trapped, the trapping laser is occluded, by electronic modulation of the trapping laser, to allow the particle to diffuse by Brownian motion. The occlusion is removed after approximately 100 ms and the particle returns to the focal point. This unoccluded portion of the particle's trajectory is compared with the simulation.

3.5 Results and discussion

Figure 3.4 plots simulation data against the experimental data. Here, a single simulated trajectory from each simulation is compared with the experimental data using different scaling factors, $a_2 = \{1, 0.1, 0.3, 0.01\}$. The 0 value on the displacement

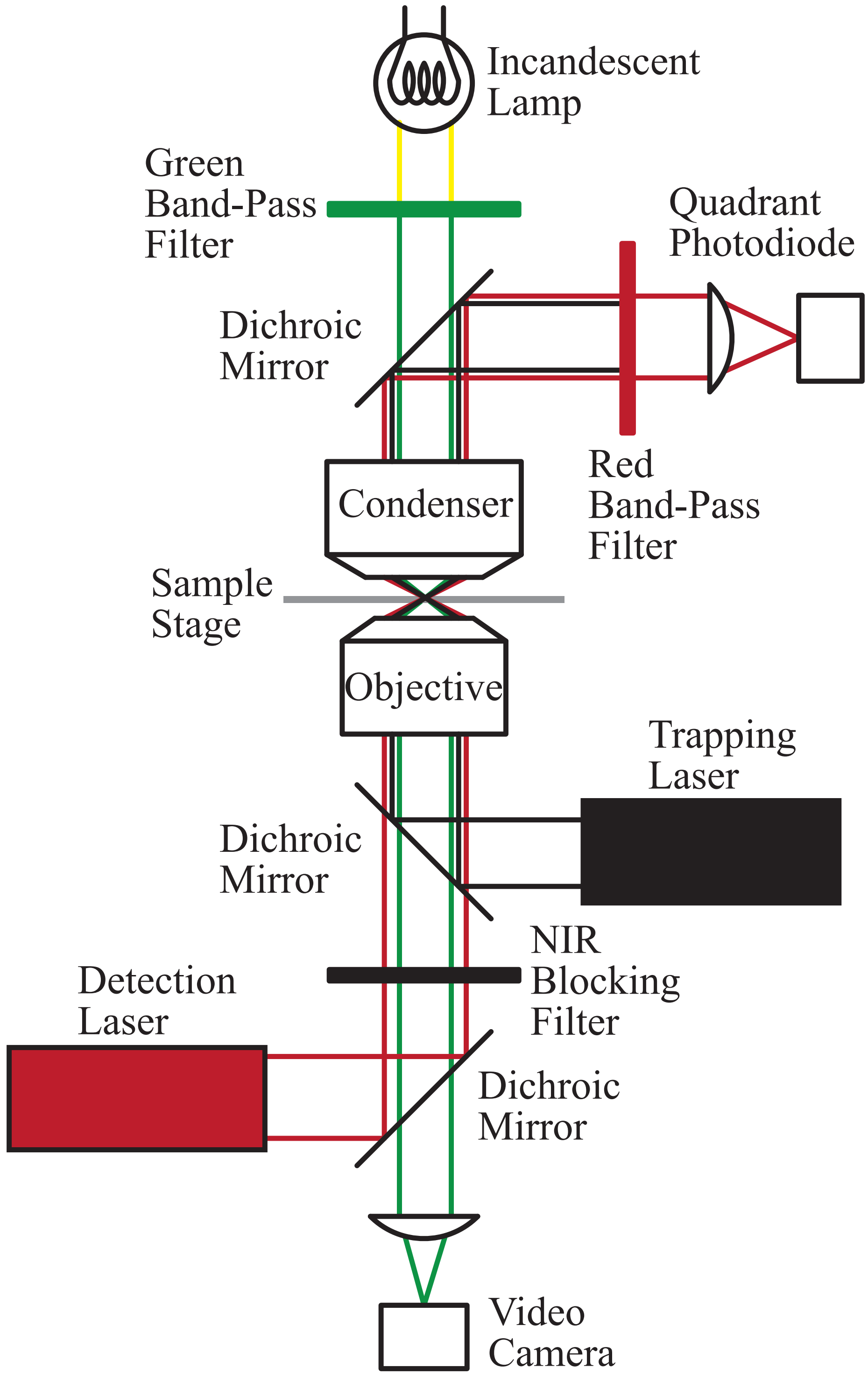


Figure 3.3: Experimental setup of the Optical Tweezer at the Los Alamos National Laboratory

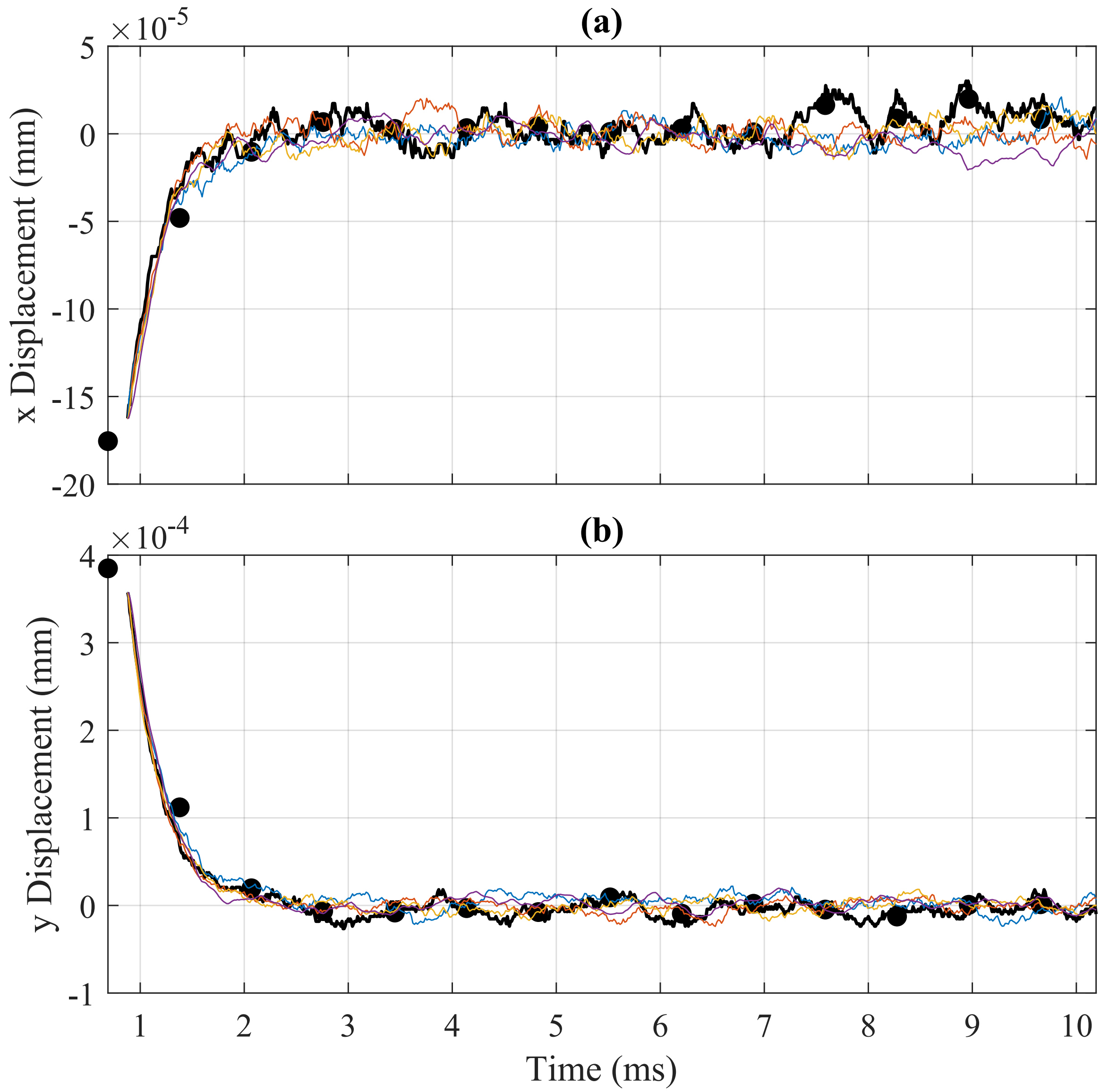


Figure 3.4: Simulation result at different scaling factors, $a_2 = 1$(—), $a_2 = 0.3$(—), $a_2 = 0.1$(—), $a_2 = 0.01$(—), and experimental data from the camera(●) and QPD(—)

axis of both Figs. 3.4a and 3.4b represent the focal point of the optical tweezer. Note that regardless of the scaling factor, the overall trajectory from each simulation matches with the experimental data. It can also be observed that the high-frequency detail in the particle's trajectory reduces as the scaling factor is decreased.

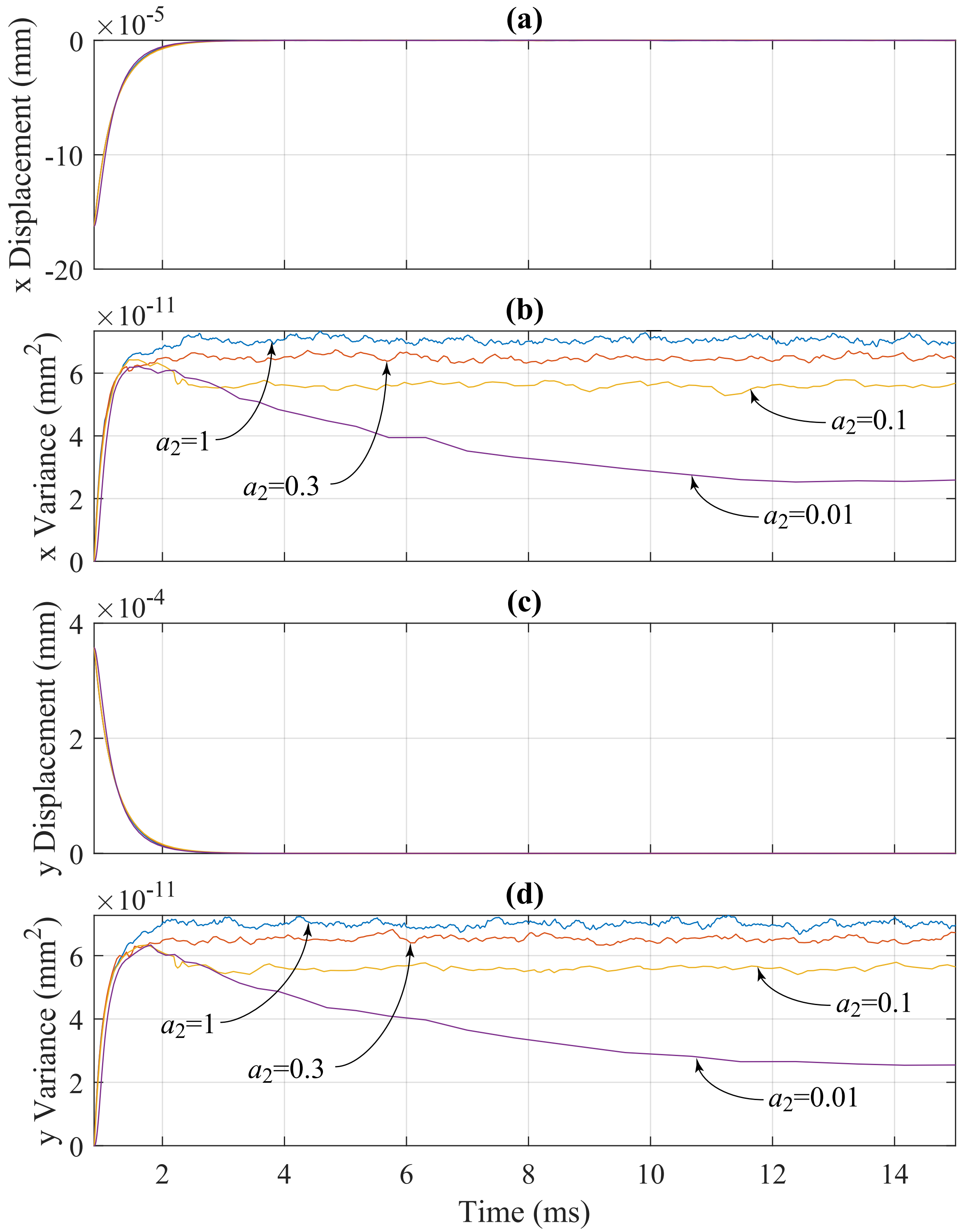


Figure 3.5: Statistics of the simulation result at different scaling factors

A comparison of different scaling factors is shown in Fig. 3.5. This comparison provides a statistical summary of 10,000 simulations. Note that as the system is scaled more, the mean trajectory and variance deviate from the unscaled system. However, this deviation is more pronounced in the variance values than in the mean trajectory. The computational time required for each solver for the different cases is given in Table 3.2. Although the SDE solvers are quite fast on their own, it is clear that the scaled system requires much less computational time than the unscaled system. However, it is important to choose a scaling factor carefully so that the final solution does not deviate much from the unscaled system. Also, note that some solvers did not converge (DNC) to a solution at a smaller scaling factor. The choice of an appropriate scaling factor is dependent on the PSD analysis which, is discussed in the next chapter.

Method	Scaling Factor			
	1	0.3	0.1	0.01
ImplicitEM	163.42	79.38(51.4%)	47.54(70.9%)	24.48(85.0%)
ImplicitEulerHeun	184.55	80.46(56.4%)	47.74(74.1%)	25.41(86.2%)
ImplicitRKMil	131.95	58.78(55.4%)	35.41(73.1%)	18.03(86.3%)
ISSEM	204.87	96.37(52.9%)	55.41(72.9%)	DNC
ISSEulerHeun	204.27	97.36(52.3%)	55.61(72.7%)	DNC

Table 3.2: Computational time in seconds, and percent decrease in computation time as compared to the unscaled system for different adaptive SDE solvers with different scaling factors

3.6 Conclusion

A novel scaling approach based on the method of multiple scales was discussed in this work, which reduces the computational time required to solve stiff stochastic differential equations using modern adaptive SDE solvers. An example of a 2μm silica particle in an optical tweezer was used to present the effectiveness of the proposed approach. It was also shown that even though the scaling approach changes the differential equations of the system and reduces the high-frequency oscillation, the solution remains in the close vicinity of the experimental data. The proposed approach required 82.47sec of computation time on average for the scaling factor of 0.3, which is more than twice as fast as the computation time required to solve the unscaled system.

CHAPTER 4

UNDERDAMPED MOTION AND PSD ANALYSIS

4.1 Introduction

The model developed in the last chapter includes forces that contribute the most to the behavior of a particle in an optical tweezer. However, the underdamped behavior observed in the old experimental data[34], presented in Section 1.4, could only be replicated using the scaled model. Furthermore, current literature [72, 63, 62, 70] also suggests that the conventional model will not predict the underdamped motion observed in the experimental data. This chapter provides a mathematical analysis and discussion of why this is the case and suggests a new experimental investigation for further analysis of this phenomenon.

A Power Spectral Density (PSD) analysis[8, 9] is commonly used for calibrating the optical tweezers, i.e. computing the trap stiffness. This chapter presents a theoretical background on the PSD analysis and compares the PSD of the scaled and the unscaled system. This comparison also showed a discrepancy between the scaled and unscaled model similar to the particle trajectories[43]. The simulated PSD of the scaled model provides a different profile than previously observed[44, 90, 52]. However, the lack of data from the old experiment increases the need for new experiments to compare the PSD profile. This chapter provides a discussion on the comparison and defines the requirements for future experiments.

This chapter also discusses an approach to take advantage of the PSD analysis to estimate the value of the scaling factor without the use of experimental data. However, this approach relies on keeping the PSD profile close to the one given by the unscaled model. Thus, the underdamped behavior is not shown by this approach unless the unscaled model shows the underdamped behavior.

4.2 Investigation of the underdamped motion

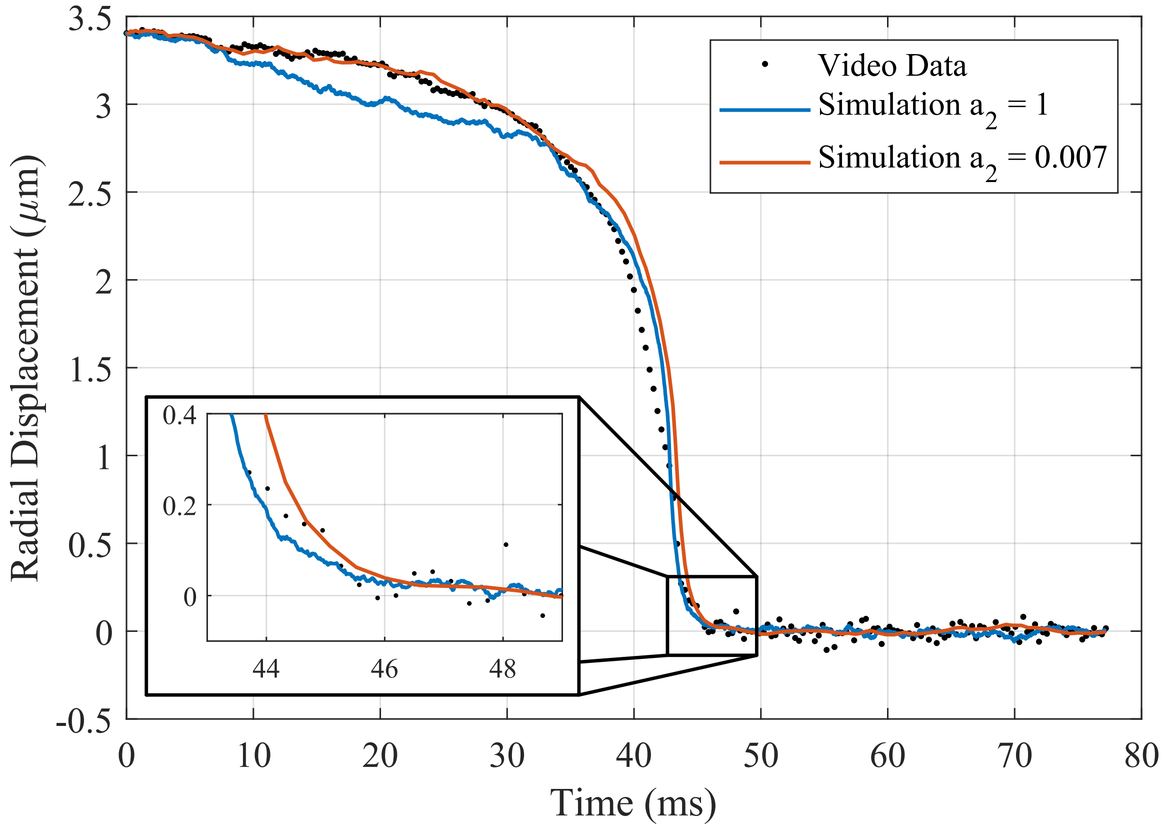


Figure 4.1: Comparison of experimental, simulated unscaled, and simulated scaled trajectory of 1950nm diameter particle

A comparison of the experimental data with the simulated trajectory from the unscaled model and the scaled model for 1950nm, 990nm and 500nm diameters particles is shown in Figs. 4.1, 4.2 and 4.3 respectively. Note that there is an observable overshoot of $\approx$ 100nm in the experimental trajectory of the 500nm diameter particle. Such an overshoot is not visible in the trajectory of the 1950nm diameter particle.

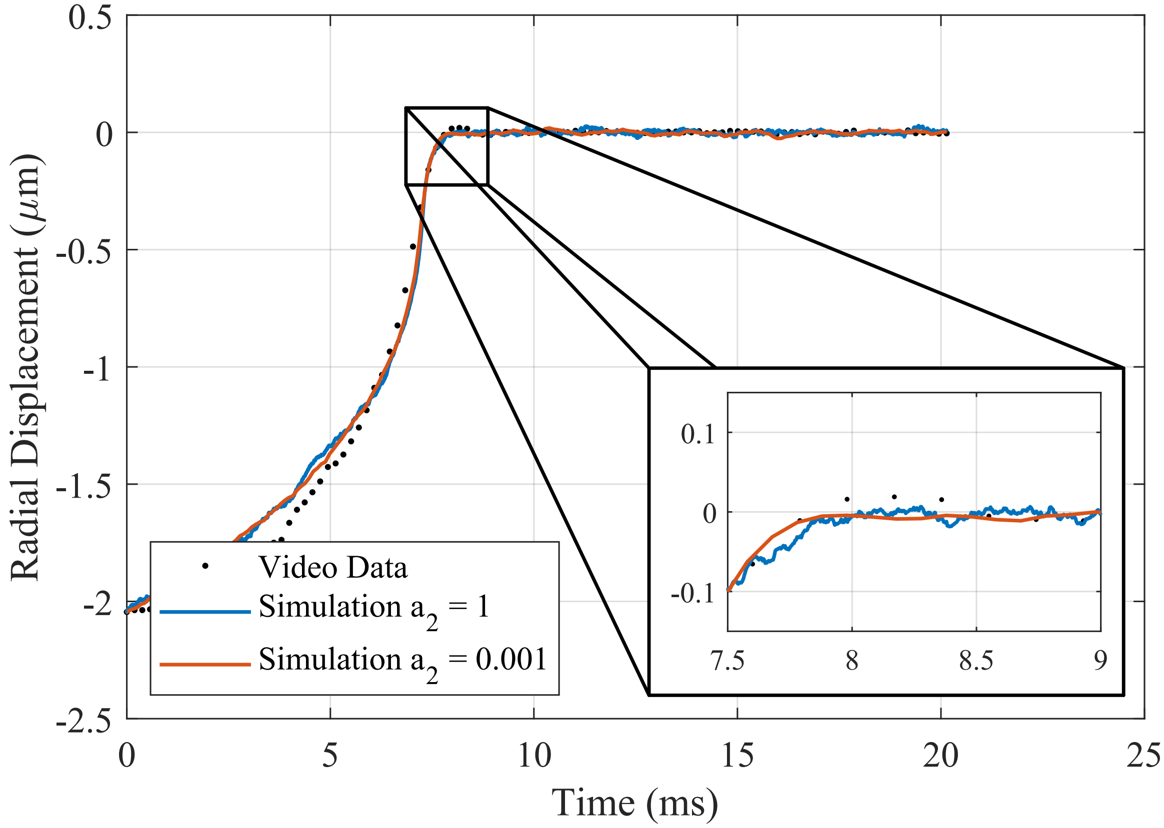


Figure 4.2: Comparison of experimental, simulated unscaled, and simulated scaled trajectory of 990nm diameter particle

Moreover, the simulated trajectory of the unscaled model does not have a noticeable overshoot near the focal point for any particle size.

The lack of underdamped motion in the unscaled model can be explained by analyzing a simplified version of the particle's EOMs given in Eqn. (3.11). Note that Eqn. (3.11) is similar to a spring-mass-damper system with a step input if the Brownian motion forces are ignored. Thus, a rearranged version of Eqn. (3.11) can be given as

$$m\ddot{\mathbf{q}} + 6\pi\mu_m r\dot{\mathbf{q}} + K\mathbf{q} = (m - \rho_m V)\mathbf{g} \tag{4.1}$$

Assuming that there is a 1.95μm diameter polystyrene ($\rho = 1060kg/m^3$) particle in the optical trap surrounded by water, the mass of the particle would be $m \approx 4.1154 \times$

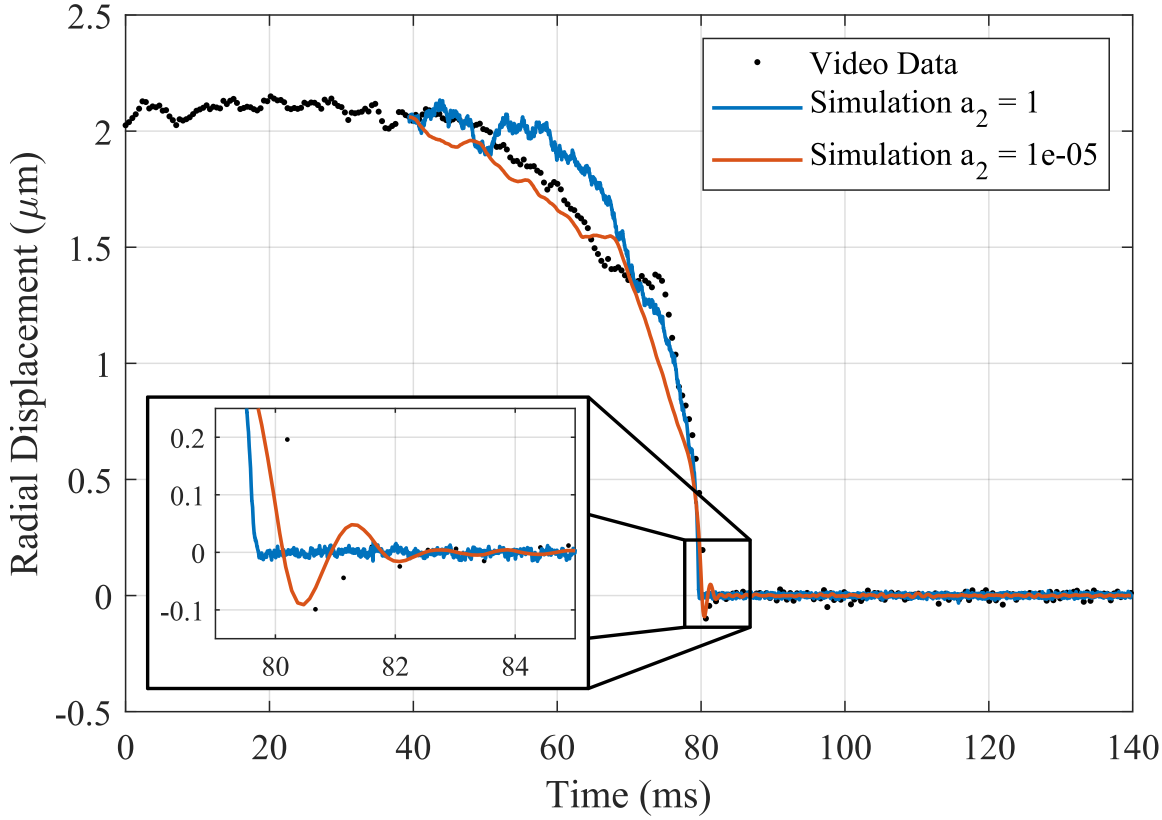


Figure 4.3: Comparison of experimental, simulated unscaled, and simulated scaled trajectory of 500nm diameter particle

$10^{-15}kg$ and the damping coefficient would be $\beta = 1.8415 \times 10^{-8} N \cdot s/m$. These values can help us estimate the trap stiffness needed to achieve the underdamped behavior using the equation of damping ratio, given as

$$\zeta = \frac{\beta}{2\sqrt{km}} \leq 1 \tag{4.2}$$

$$\therefore k \geq \frac{\beta^2}{4m} = 2.0601 \times 10^{-2} N/m = 2.0601 \times 10^4 \text{pN}/\mu\text{m} \tag{4.3}$$

The calculation above shows that the trap stiffness must be larger than $2.0601 \times 10^4 \text{pN}/\mu\text{m}$ for a $1.95\mu\text{m}$ diameter particle to show a visible underdamped behavior. This situation becomes worse as the particle size reduces. For instance, the trap stiffness will have to be at least $8.0342 \times 10^4 \text{pN}/\mu\text{m}$ for a 500nm polystyrene particle to show an underdamped behavior. The stiffness achieved by an optical tweezer for such an experiment is typically limited to 1000pN/μm [75, 87, 59, 79]. Thus, it is clear that the unscaled model should have an overdamped response near the focal point as per this theoretical analysis.

A similar analysis can be performed for the scaled model, Eqn. (3.21). The rearranged scaled EOMs for a particle can be given as

$$m\ddot{\mathbf{q}} + a_2 6\pi\mu_m r\dot{\mathbf{q}} + a_2 K\mathbf{q} = a_2(m - \rho_m V)\mathbf{g} \tag{4.4}$$

Here, the value of the scaling factor, a_2, is chosen to match the simulated trajectory with the experimental data. Table 4.1 gives the scaling factors for different particle sizes. Values for these scaling factors can be used to estimate the trap stiffness required, from Eqn. (4.2), for the particles to show the underdamped motion. Table 4.1 also shows these trap stiffnesses, the trap stiffness required for the unscaled model to show the underdamped motion and the estimated experimental trap stiffnesses[42]. Note that the stiffness required to show an underdamped motion is much lower in the case of the scaled model as compared to the unscaled model.

Particle Diameter (nm)	1950nm	990nm	500nm
Scaling Factor	7×10^{-3}	1×10^{-3}	1×10^{-5}
Experimental Stiffness (pN/μm)	61.54	133.34	200
Unscaled model Stiffness (pN/μm) @$\zeta = 1$	2.0601×10^4	4.0577×10^4	8.0342×10^4
Scaled model Stiffness (pN/μm) @$\zeta = 1$	1.4420×10^2	4.0577×10^1	8.0342×10^{-1}

Table 4.1: Trap stiffness comparison between the experiment, unscaled model and scaled model for underdamped motion of particles with different sizes

Another version of such analysis can be presented from a fluid dynamics perspective. The combination of the particle sizes and the fluid environment of the optical trapping experiment yields a situation that can be characterized by a low particle Reynolds number ($\mathrm{Re_p} < 1$). The $\mathrm{Re_p}$ is meant to characterize the properties of the fluid based on the motion of an object moving through it. Particularly, it characterizes the fluid viscosity in relation to the terminal velocity of an object moving through the still fluid. The $\mathrm{Re_p}$ is defined as,

$$\mathrm{Re_p} = \frac{\rho_m s L}{\mu_m} = \frac{sL}{\nu_m} = \frac{\text{Inertial Forces}}{\text{Viscous Forces}} \tag{4.5}$$

where, ρ_m is the fluid's density, s is the maximum relative velocity between an object and the fluid, L is a characteristic length (the diameter of the particle here), μ_m is the dynamic viscosity of the fluid, and ν_m is the kinematic viscosity of the fluid. If the fluid is not moving or still, s is the object's terminal velocity. The $\mathrm{Re_p}$ is interpreted as representing the relative importance between inertial and viscous forces. Thus, a small Reynolds number suggests that the inertia forces would be smaller than the viscous damping. The $\mathrm{Re_p}$ for all of the experiments are small, $\mathrm{Re_p} < 10^{-2}$, meaning that all the particles should show an overdamped motion.

The $\mathrm{Re_p}$ over time for the differently sized particles are shown in Figs. 4.4 through 4.6. The $\mathrm{Re_p}$ for the 990nm particle is smaller than or equal to 1×10^{-3}, see Fig. 4.5. These small values of the $\mathrm{Re_p}$ should indicate overdamped motion, but the previous experimental results contradict this conclusion. Even more telling are the results for the 500nm particle in Fig. 4.6 where $\mathrm{Re_p} \leq 3 \times 10^{-4}$. This should indicate an even greater inclination toward the overdamped motion of the 500nm particle, similar to what was discussed through the damping ratio calculation near Eqn. (4.2). However, the previous experimental data in Fig. 4.3 indicate just the opposite, a greater inclination toward underdamped motion. This is a significant contradiction for the predictions of a low $\mathrm{Re_p}$.

The discrepancy between the experimental data and the trajectories from the unscaled model suggests that the model is either missing some forces or the force

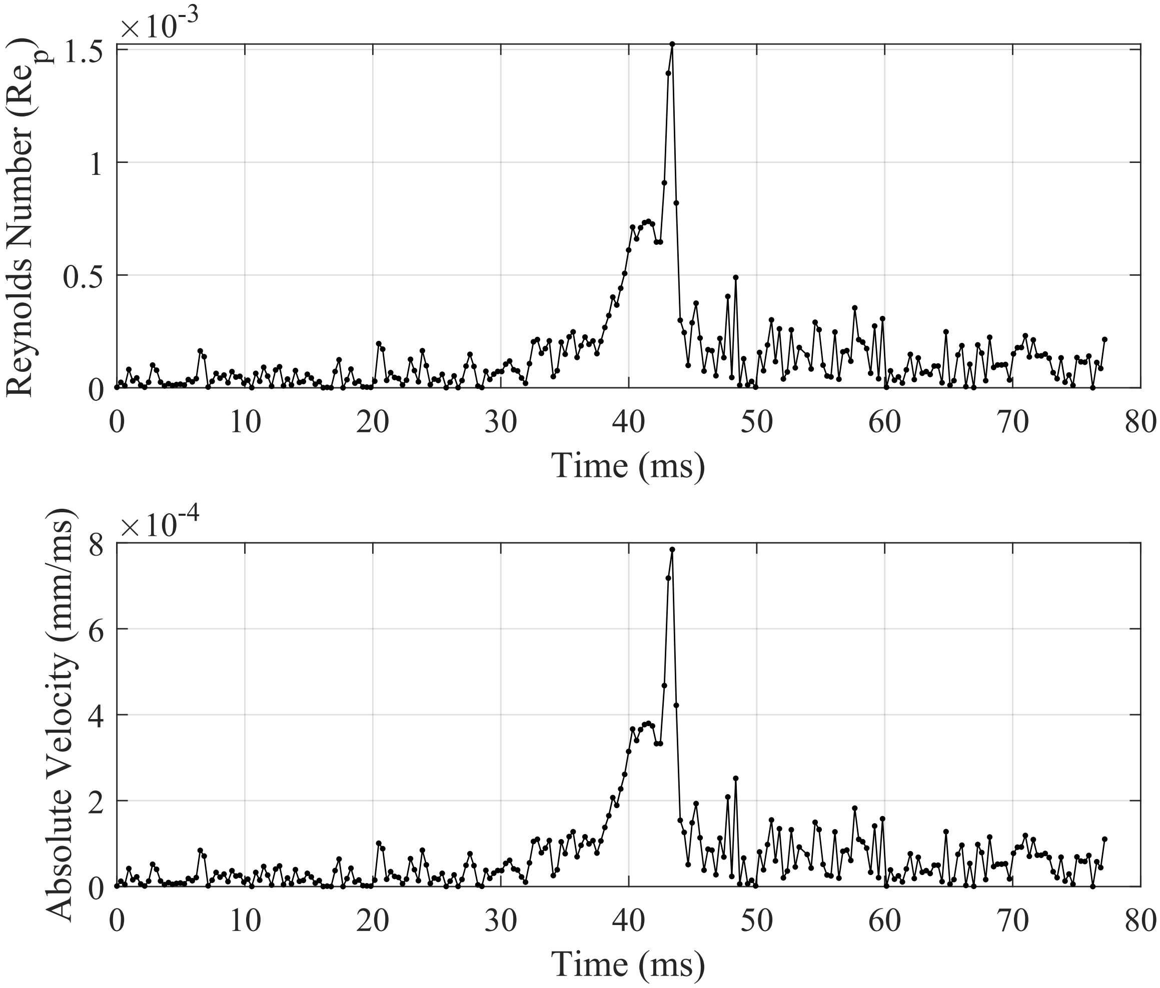


Figure 4.4: Particle Reynolds number (Re_p) and absolute velocity for 1950nm diameter particle

models used are not accurate. The match achieved between the simulation and the experiment by the use of the scaled model strengthens the hypothesis that the force models used for damping, laser beam or both might be inaccurate. However, it was difficult to pinpoint the source of this discrepancy due to the low time resolution and the lack of repeatable experimental data. Thus, it was necessary to do the experiments again to collect new high-resolution, reliable and repeatable data.

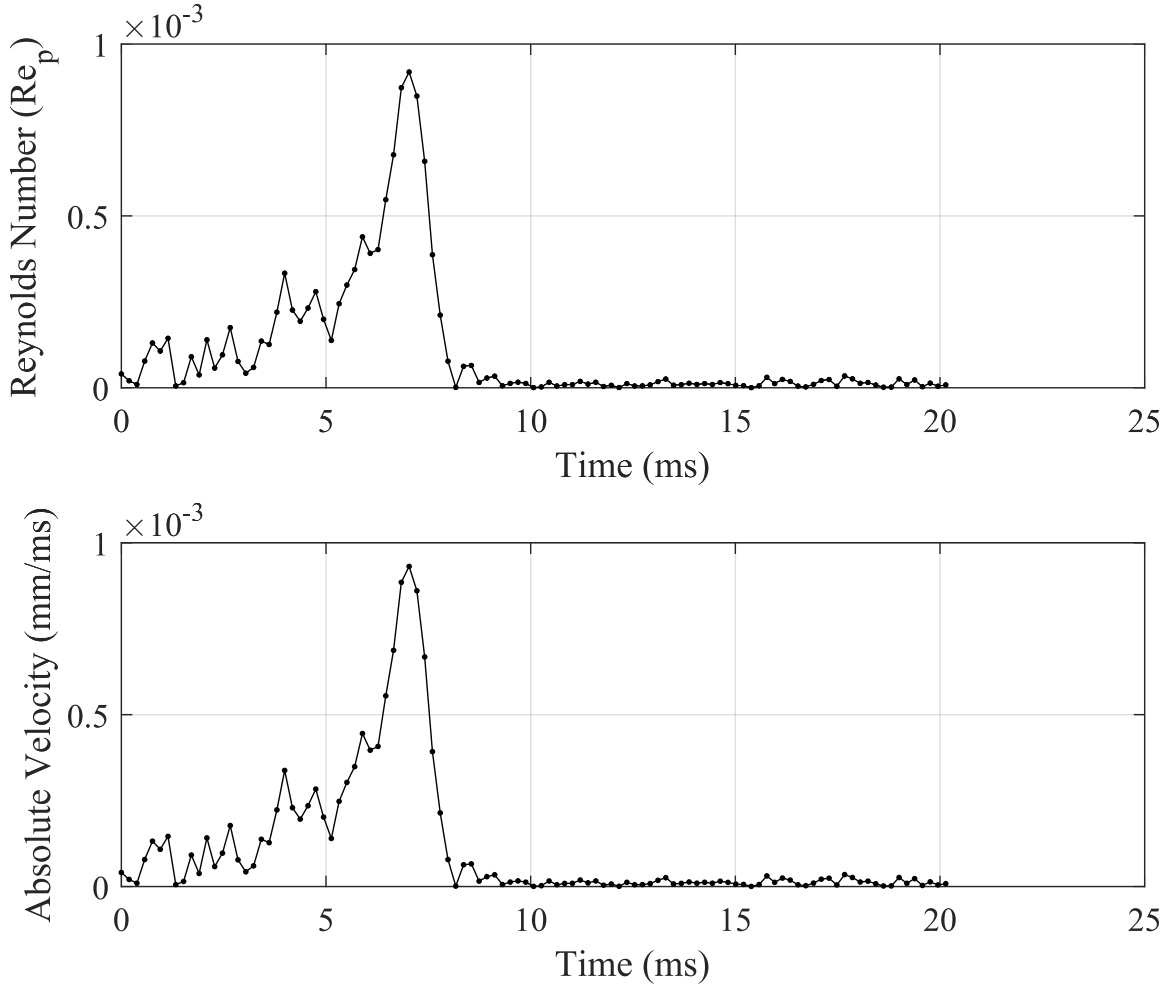


Figure 4.5: Particle Reynolds number (Re_p) and absolute velocity for 990nm diameter particle

4.3 PSD Analysis

A particle trapped at the focal point of an optical trap has different Brownian motion characteristics as compared to a particle freely moving in a fluid. This is mainly due to the external laser forces acting on the particle. The power spectral density analysis takes advantage of this property to estimate the stiffness of the optical

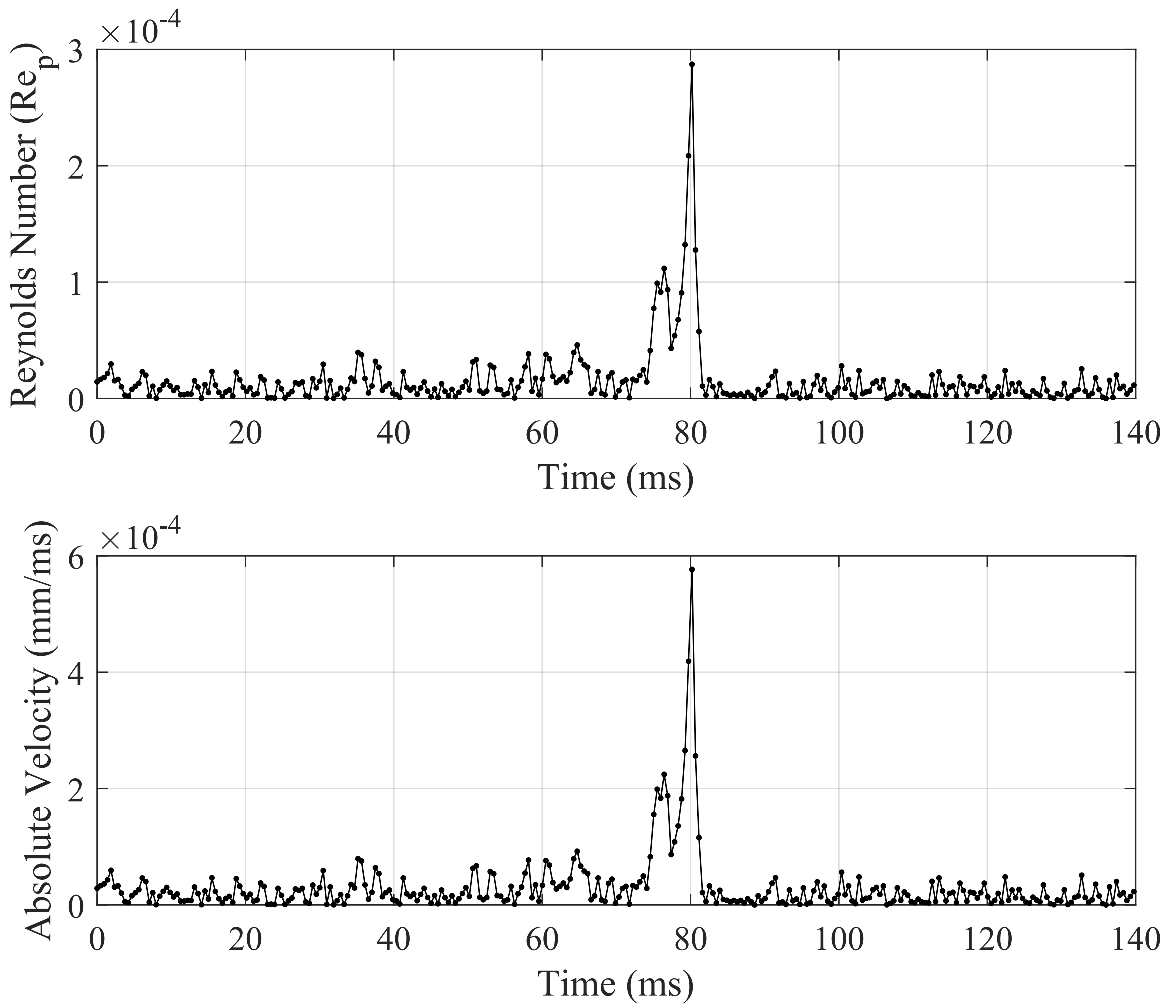


Figure 4.6: Particle Reynolds number (Re_p) and absolute velocity for 500nm diameter particle

trap experimentally [8, 9]. For this analysis, let's consider the one-dimensional version of Eqn. (3.11) given as

$$m\ddot{q}_x + \beta\dot{q}_x + k_{xx}q_x = \eta(t)\sqrt{2k_BT\beta} \tag{4.6}$$

where, m is the mass of the particle, β is the Stokes drag coefficient, k_{xx} is the optical trap stiffness in x-direction, k_B is the Boltzmann constant, T is the system temperature and $\eta(t)$ is a random Gaussian process as discussed in Section 3.2.4.

Note that the gravity and buoyancy forces are ignored here since those are smaller compared to the laser forces.

The power spectral density can be defined in three different equivalent ways. Here, the definition via finite Fourier transforms[7] is used since it will be directly applicable to both the experimental and simulated data. The power spectral density, for a recording of a particle's position over a time period T_s, can be given as

$$P(f) = \frac{2}{T_s} \left|\tilde{q}_x(f)\right|^2 \tag{4.7}$$

where $\tilde{q}_x(f)$ is the Fourier transform of the particle's position in the x-direction. This can be computed by calculating the Fourier transform of Eqn. (4.6) as shown below,

$$m(i2\pi f)^2\tilde{q}_x(f) + \beta(i2\pi f)\tilde{q}_x(f) + k_{xx}\tilde{q}_x(f) = \tilde{\eta}(f)\sqrt{2k_BT\beta} \tag{4.8}$$

$$\therefore \tilde{q}_x(f) = \frac{\sqrt{2k_BT\beta}}{k_{xx} - 4m\pi^2 f^2 + 2i\beta\pi f}\tilde{\eta}(f) = \frac{\sqrt{D/2\pi^2}}{f_c - 2\pi\varepsilon f^2 + if}\tilde{\eta}(f) \tag{4.9}$$

where, $D = k_BT/\beta$ is known as Einstein's diffusion constant, $f_c = k_{xx}/2\pi\beta$ is known as the corner frequency, $\varepsilon = m/\beta$ as defined in Secc. 3.3 and $\tilde{\eta}(f)$ is the Fourier transform of the Gaussian white noise. The absolute square value of $\tilde{q}_x(f)$ can be computed by multiplying it with its complex conjugate, $\tilde{q}_x^\star(f)$, as shown below

$$\left|\tilde{q}_x(f)\right|^2 = \frac{\sqrt{D/2\pi^2}}{f_c - 2\pi\varepsilon f^2 + if} \cdot \frac{\sqrt{D/2\pi^2}}{f_c - 2\pi\varepsilon f^2 - if}\tilde{\eta}(f)\tilde{\eta}^\star(f) \tag{4.10}$$

$$\therefore \left|\tilde{q}_x(f)\right|^2 = \frac{D/2\pi^2}{f_c^2 + (1 - 4\pi\varepsilon f_c)\, f^2 + 4\pi^2\varepsilon^2 f^4} \left|\tilde{\eta}(f)\right|^2 \tag{4.11}$$

The power spectral density of Gaussian white noise is 1[7]. Thus, the value of $\left|\tilde{\eta}(f)\right|^2$, using the definition of the power spectral density, would be $\left|\tilde{\eta}(f)\right|^2 = T_s/2$. The power spectral density of the particle's position can now be defined as,

$$P(f) = \frac{2}{T_s} \left|\tilde{q}_x(f)\right|^2 = \frac{D/2\pi^2}{f_c^2 + (1 - 4\pi\varepsilon f_c)\, f^2 + 4\pi^2\varepsilon^2 f^4} \tag{4.12}$$

The trap stiffness from the experiment can thus be calculated by fitting the equation above to the PSD of the particle's position, measured using a high-speed camera or a quadrant photodiode. This allows the value of fc to be estimated, which directly relates to the trap stiffness. Equations similar to (4.12) including different models for fluid forces are available in [8, 9]. The MATLAB code for computing the PSD from experimental data is provided in Appn. J.2.1 and the code for fitting Eqn. (4.12) to the PSD generated from the experimental data is provided in Appn. J.2.5.

Note that the equation of the PSD of the particle's position, Eqn. (4.12), is for the unscaled model. Thus, for the scaled equation of motion,

$$m\ddot{q}_x + a_2\beta\dot{q}_x + a_2k_{xx}q_x = a_2\eta(t)\sqrt{2k_BT\beta} \tag{4.13}$$

the equation of the PSD can be given as

$$P(f) = \frac{2}{T_s}\left|\tilde{q}_x(f)\right|^2 = \frac{D/2\pi^2}{f_c^2 + \left(1 - \dfrac{4\pi\varepsilon f_c}{a_2}\right)f^2 + \dfrac{4\pi^2\varepsilon^2}{a_2^2}f^4} \tag{4.14}$$

Figure 4.7 shows the equation above plotted with different scaling factors for 2μm particle diameter. Note that amplitudes at higher frequencies decrease as the scaling factor is decreased. This was visible in the simulation plots discussed in Section 3.5. However, decreasing the scaling factor decreases the computation time as well, as was shown in Table 3.2, which provides the incentive to reduce the scaling factor.

4.4 Choosing a Scaling Factor

From the EOMs of the particle, note that the sampling rate should be on the order of the relaxation time, i.e. $\tau = m/\beta = 4.4356 \times 10^{-7}$sec for a 2μm diameter particle, to observe the transients in the particle's velocity. The sampling rate for the experimental data collected using a camera is generally limited to ≈ 25kHz, or to ≈ 100kHz if a quadrant photodiode is used. Thus, the velocity transients will not be

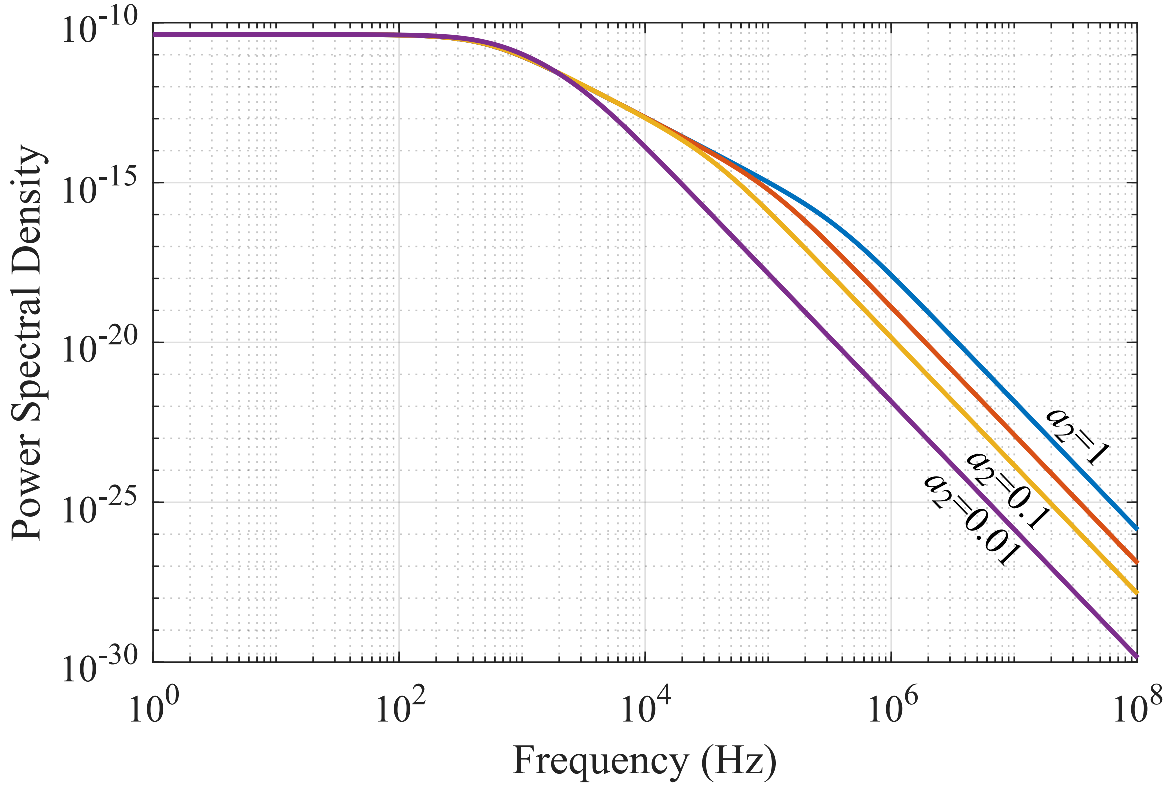


Figure 4.7: Theoretical PSD at different scaling factors for 2μm particle (the unlabeled line is $a_2 = 0.3$)

visible in the experimental trajectories. This information can be used to choose an appropriate value of the scaling factor.

The trends shown in Figure 4.7, i.e. the effect of scaling the EOMs on the PSD of the particle's position, can be observed in the simulation data given in Figure 4.8 as well. To provide a baseline for comparison, the PSD from the theoretical Eqn. (4.12) for the unscaled system, $a_2 = 1$, and that of the experimental data are given in Figure 4.8a. Note that the data for experimental PSD has frequency values up to 50kHz only, which is half of the sampling frequency of the QPD at 100kHz. It can also be observed that the experimental PSD deviates from the theoretical plot at higher frequencies. This is attributed to the sensor noise. Rest of the subplots in Figure 4.8, i.e. Figure 4.8b to Figure 4.8e, compares the PSD from the theoretical

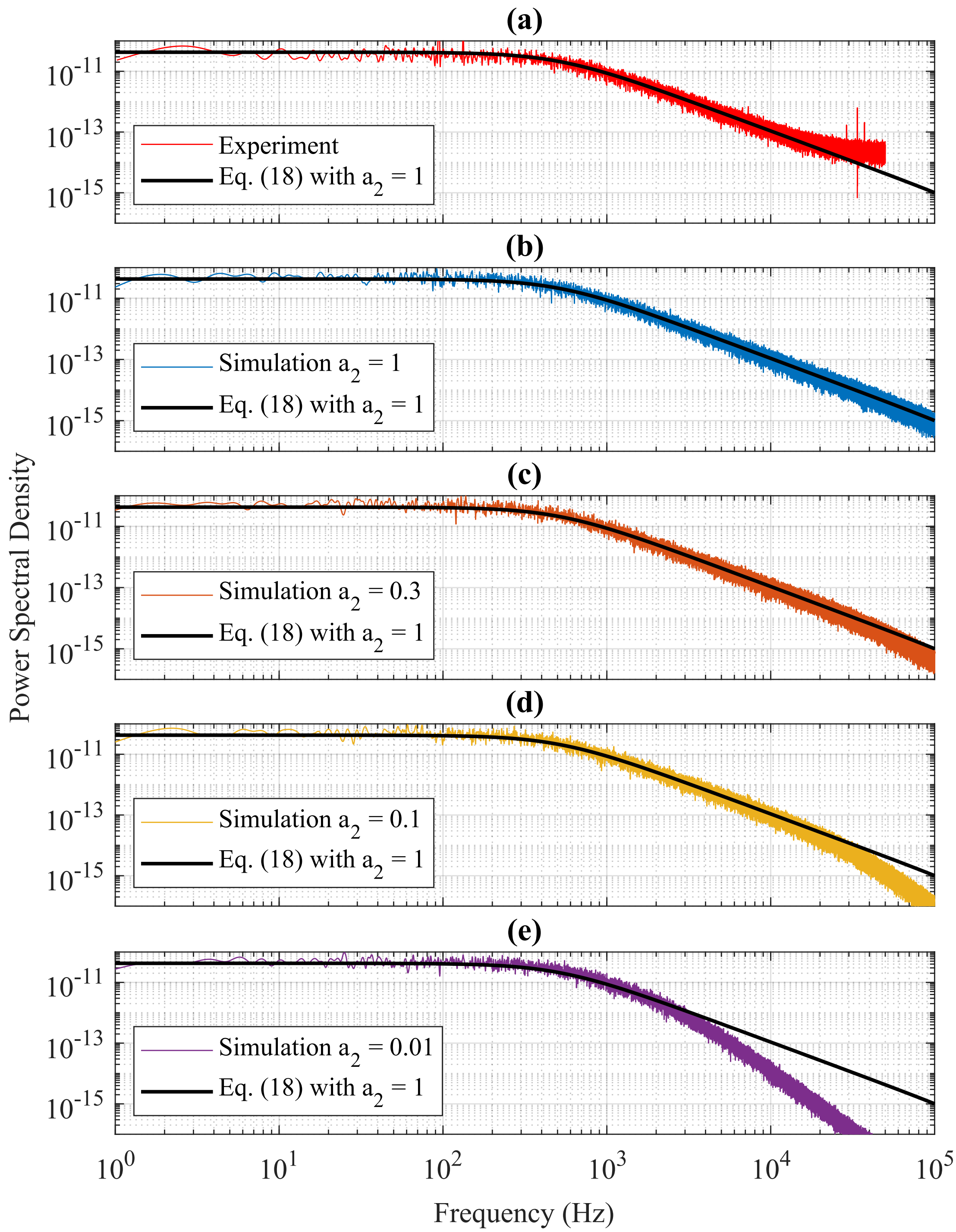


Figure 4.8: Comparison of theoretical PSD of unscaled system with experimental PSD and simulation PSD with different scaling factors

Eqn. (4.12) for the unscaled system and the PSD obtained from the system simulated with different scaling factors.

The PSD obtained from the simulation data for $a_2 = 0.3$, Figure 4.8c, provides the closest match to the theoretical PSD of the unscaled system throughout the entire frequency range, $\approx$ 100kHz. The PSD computed from simulation data loses high-frequency information as the scaling factor is decreased, as suggested in Figure 4.7. However, all the scaling factors yield similar corner frequency values, f_c, which is a direct measure of the trap stiffness.

Note that if the simulation data in Figure 4.8 is input to a curve fitting calculation [8, 9], it will attempt to match the unscaled PSD, Eqn. (4.14) with $a_2 = 1$, to the scaled data. This may yield vastly different values for the corner frequencies in each case. This wide variation can be reduced by limiting the data for the curve fit down to a smaller frequency range, as the scaling factor is decreased, where the discrepancy between scaled and unscaled system is less according to Figure 4.7. The result of this exercise is shown in Table 4.2 where the corner frequencies remain somewhat close to the unscaled one.

Table 4.2: Corner frequency value for systems with different scaling factors

Scaling Factor	Corner Frequency	Frequency Range
1	541.2Hz	1Hz - 20kHz
0.3	525.6Hz	1Hz - 20kHz
0.1	498.6Hz	1Hz - 10kHz
0.01	478.0Hz	1Hz - 2kHz

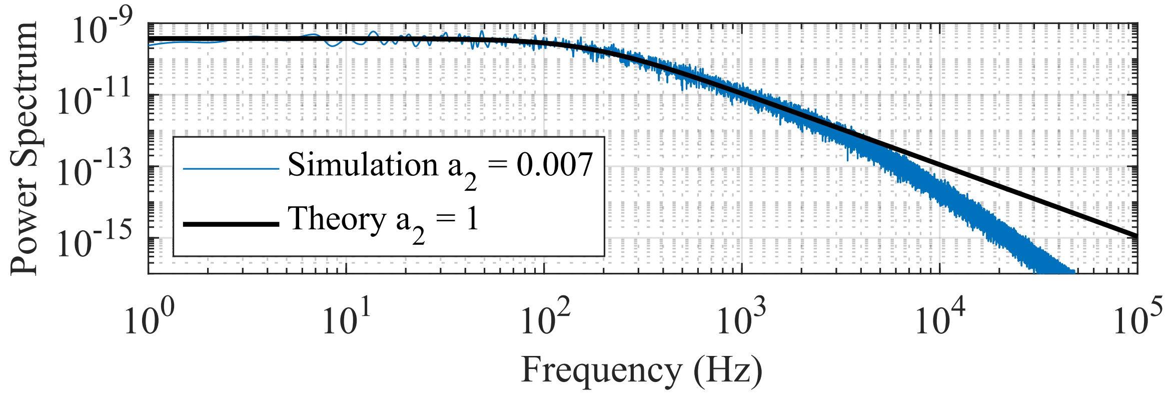


Figure 4.9: Theoretical PSD from the unscaled model vs Simulated PSD from the scaled model for 1950nm diameter particle

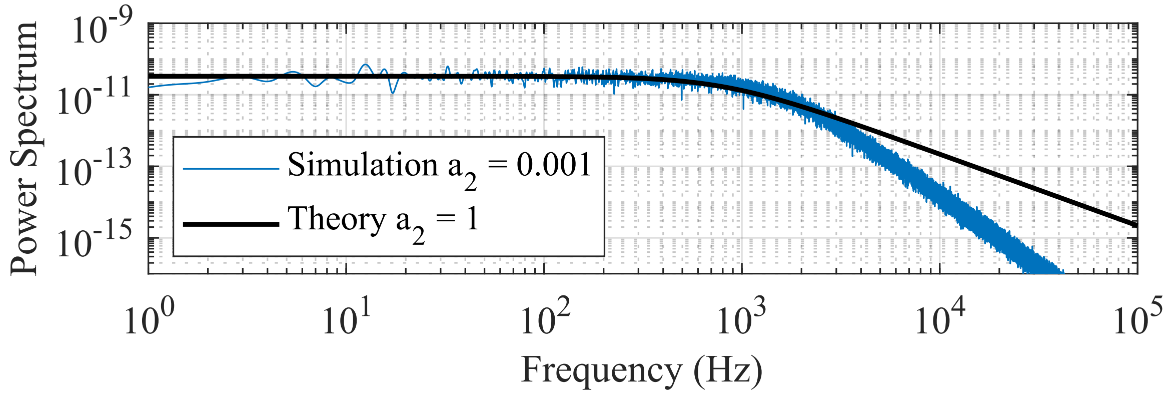


Figure 4.10: Theoretical PSD from the unscaled model vs Simulated PSD from the scaled model for 990nm diameter particle

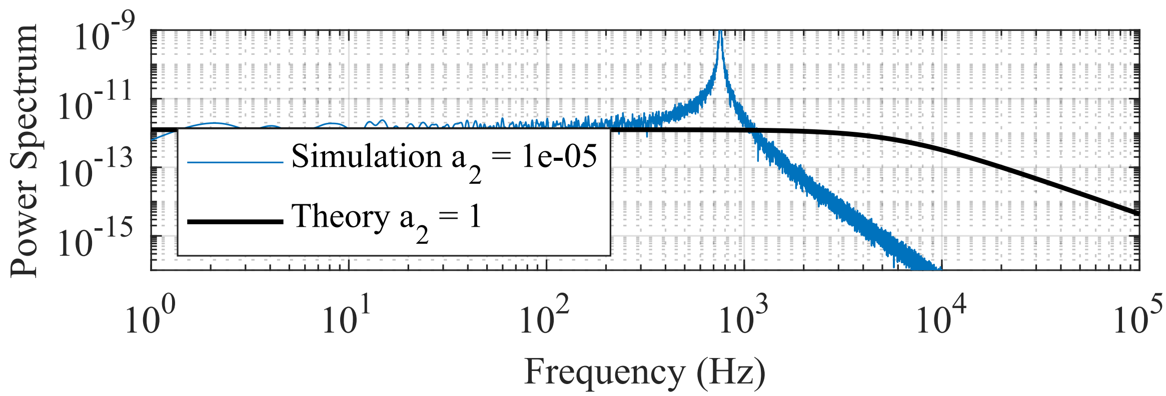


Figure 4.11: Theoretical PSD from the unscaled model vs Simulated PSD from the scaled model for 500nm diameter particle

4.5 PSD of particles from old experiments

The experimental PSD from the old experiments cannot be computed since the high temporal resolution data for a long time period is not available. Thus, the PSD for old experiments is estimated by first matching the trajectory of a simulation to the experimental trajectory and then using the corresponding scaling factor to compute the PSD with the help of the simulation. The estimated PSD for each particle size is provided in Figs. 4.9, 4.10 and 4.11. Note that for the 1.95μm diameter particle, the estimated PSD matches well with the theoretical PSD for a wide range of frequencies. However, the estimated PSD of 0.99µm and 0.5µm diameter particles differ significantly from the theoretical PSD.

The PSD estimated for the old experimental data provides another parameter that can be compared with future experimental data. It can be noted that if the particle's motion shows an underdamped behavior, then the PSD would achieve a distinct peak before it starts to reduce at higher frequencies. Such a behavior has been observed experimentally in the past [28] where the cause of the peak was attributed to hydrodynamic memory. However, this study used melamine resin particles with diameters between 2 and 3 μm and suspended them in acetone, which are different experimental parameters than being analyzed here.

The PSD profile of the scaled EOMs of the 500nm diameter particle also suggests that there might exist a discrepancy between the experimental conditions and the simulation model. Such discrepancies may include the hydrodynamic memory[28] discussed in the last paragraph, laser misalignment and aberrations [95, 76], thermal effects due to local heating [19] or other fluid effects [26]. Unfortunately, enough data is not available from old experiments for the analysis of these discrepancies. Thus, setting up a new experiment and collecting more data is necessary.

4.6 Suggestions on new Experimental Setup

Note that the possible discrepancies between the experiment and simulation model, discussed in the last section, can be classified into two broad categories corresponding to fluid or laser forces acting on the particle. Two experiments can be set up to isolate the effects of each major force and analyze them in detail.

One of the experimental setups would be an optical trap designed to minimize the effects of beam misalignment and aberrations along with a well-defined laser beam profile. This will achieve a closer match between the laser beam forces in the experiments and the simulation. Once achieved, trap-release-retrap experiments can be conducted where a trapped particle can be released and retrapped within a few milliseconds. These experiments will help investigate the underdamped motion observed in the trajectory of a 500nm particle being trapped. On the other hand, the Brownian motion of a trapped particle can be recorded for long time periods to perform PSD analysis and look for the peak in the PSD profile. This setup and new results are discussed in the next chapter.

The other experimental setup can be designed to isolate the effects of fluid forces on the particle. The eventual goal would be to make particle transport through an "L" shaped microchannel. If there are fluid forces that do not adhere to the standard Stokes drag model [24], then the particle's trajectory would be different downstream of the microchannel than expected. However, to observe this behavior, it will be important to track single particle trajectories from a known position upstream of the microchannel. This can be achieved by designing an electrical trap that relies on the dielectrophoretic (DEP) force[51] generated by a strong oscillating electric field. This experimental setup and the results are part of future work and thus will not be discussed here.

4.7 Conclusion

The existence of the underdamped motion was investigated here for both the unscaled and scaled equations of motion. It was shown why the current unscaled model would never predict the underdamped motion for the experimental scenario being analyzed. A power spectral density analysis, commonly used to calibrate optical tweezers, was introduced, Eqn. (4.14) and Fig. 4.7, which concretely showed that decreasing the scaling factor of a stochastic differential equation model increases the loss of high-frequency content in the resulting motion. The idea of choosing an appropriate scaling factor, based on the desired preservation of high-frequency details in the PSD profile, was proposed. An example of a 2μm silica particle in an optical tweezer was used to present the effectiveness of this idea. PSD profiles for the old experimental data were generated, where the scaling factor was chosen by matching the simulated and experimental particle trajectories. It was shown that there might exist a peak in the PSD profile of the particle that showed the underdamped motion.

Based on these analyses, possible discrepancies between the simulation model and the experiment were identified. Two experimental setups were suggested to isolate the effects of different forces acting on the particle so that the suspected underdamped motion could be analyzed in further detail.

CHAPTER 5

EXPERIMENTAL EXPLORATION OF THE UNDERDAMPED MOTION

5.1 Introduction

The new optical trapping setup developed to further analyze the suspected underdamped motion is discussed in this chapter. Note that the goal is to minimize the laser aberration in the optical trap. Another goal is to keep the experimental parameters, like the trap stiffness, particle material, NA of the objective, etc. similar to the previous experiment.

This chapter also discusses the experimental procedures that were followed. The simulation can compute the correct laser forces only if the laser beam profile, used in the simulation, matches the one in the experiment. The procedures provided here were designed to achieve the best laser beam alignment and profile measurement for every experiment.

The data collected from the experiment is huge and in the form of images. Thus, all the data needs to be compressed for better storage and processed to extract the position of the particle. This chapter discusses different techniques of data compression and image processing and also touches upon the advantages and disadvantages of each.

Finally, the new experimental data for 2μm, 1μm and 500nm diameter particles is presented in this chapter. Data for both types of experiments, the Brownian motion of a trapped particle (referred to as the Brownian motion experiment) and the trap-release-retrap (referred to as the TRR experiment), is discussed here.

5.2 Experimental Setup

A simplified version of the optical tweezer platform along with the imaging setup is shown in Fig. 5.1. A continuous wave DPSS laser, Excelsior from Spectra-Physics Inc., beam with 1064nm wavelength and 800mW power was used to trap such particles. This laser was chosen because of its ability to produce an intensity profile that is close to Gaussian, which is commonly characterized by the beam quality factor (M^2) being close to 1 for TEM_{00} modes. The output beam from the laser unit has a diameter of 0.45 ± 0.05mm. However, note that a Gaussian function spans to $\pm\infty$. Thus, the beam diameter is defined as 1.7 times the full width at half maximum. This is discussed further in the next section.

A shutter was developed in-house to block the laser at desired times. The goal was to achieve low closing/opening timing. This was achieved by repurposing a dead hard disk drive (HDD). Hard drives contain a voice coil (actuator) that moves at the speed of $\approx 2900°$/s. This actuator was powered through a DRV8871 DC Motor Driver Breakout Board, from Adafruit, operating at 12V supply. The shutter was created by drilling a hole in the HDD case and attaching a razor blade to the actuator arm that would cover the hole. This design was inspired by [56] and achieved 8ms of closing/opening time after proper tuning.

The 0.45 ± 0.05mm diameter beam from the laser was expanded to ≈ 4.5mm using a 10X Galilean-type beam expander from Thorlabs. The output from the beam expander was passed through absorptive neutral density (ND) filters to reduce the laser power going toward the sample. The lower power beam is guided toward a dichroic mirror inside an inverted optical microscope, Nikon Eclipse E-800, via two 45° Broadband Dielectric Mirrors. The mirrors are placed on height-adjustable posts with 2D kinematic mounts. This setup allows four-dimensional, two position and two rotation, adjustments of the laser beam. After reflecting from the dichroic mirror, the

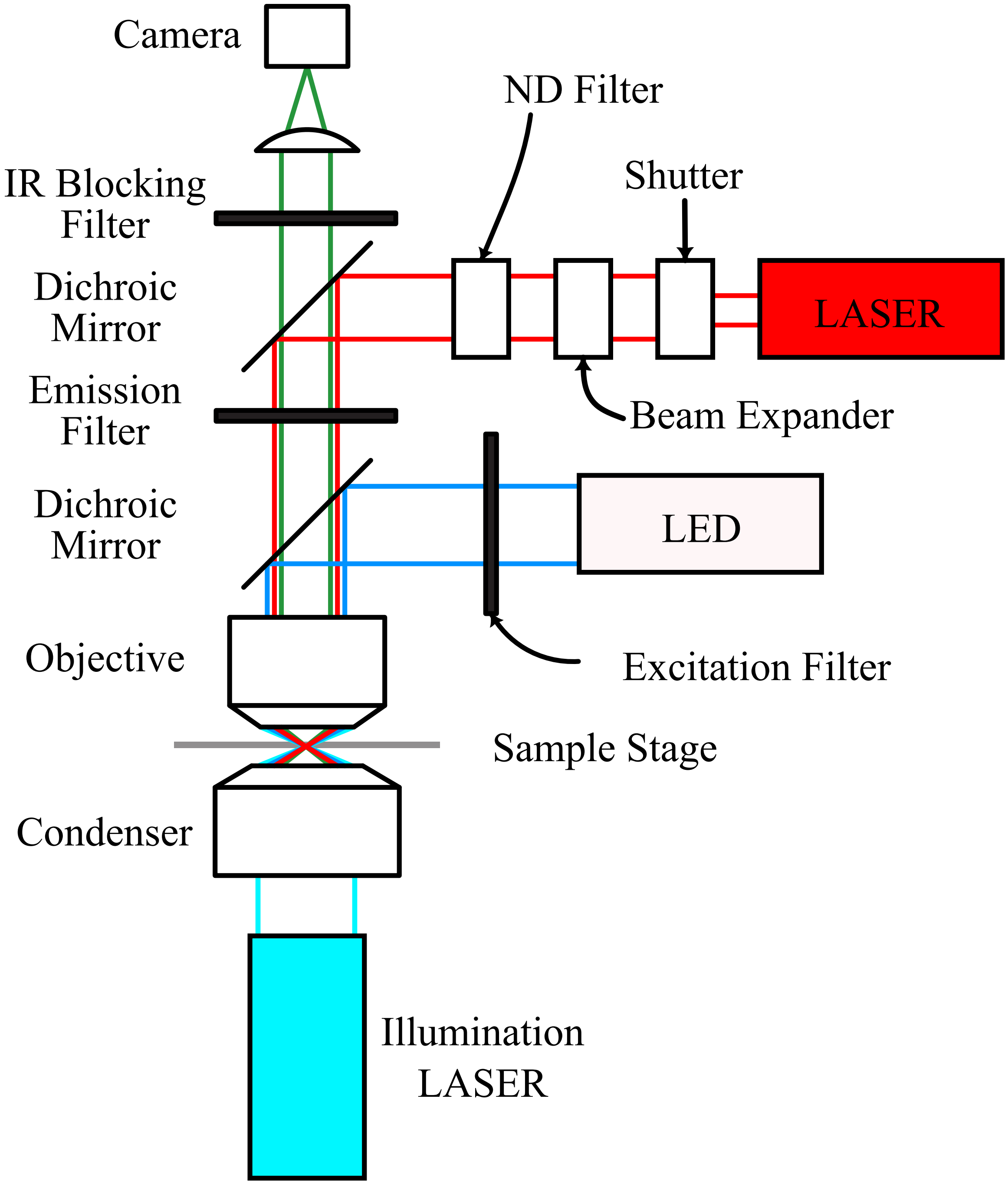


Figure 5.1: A simplified version of the experimental setup developed at the University of Texas at Arlington

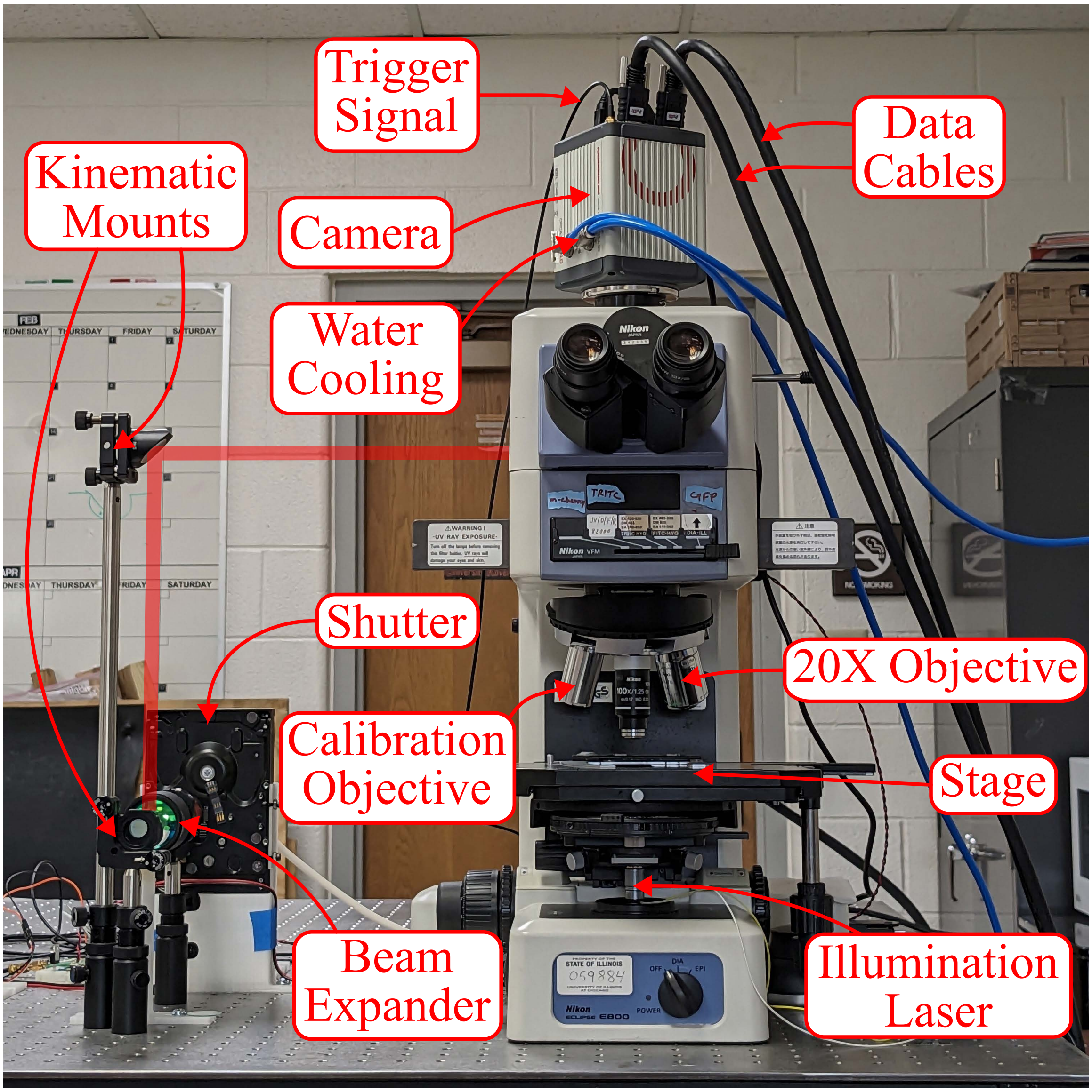


Figure 5.2: Front view of the experimental setup showing different components along with the laser path

beam enters at the back aperture of a 100x oil immersion objective, from Nikon[66], with a numerical aperture (NA) of 1.25.

The choice of the beam expander relies on the beam diameter at the output of the laser and the size of the back aperture of the objective lens. In this case, the back aperture of the objective lens is 7mm in diameter. This is important because if

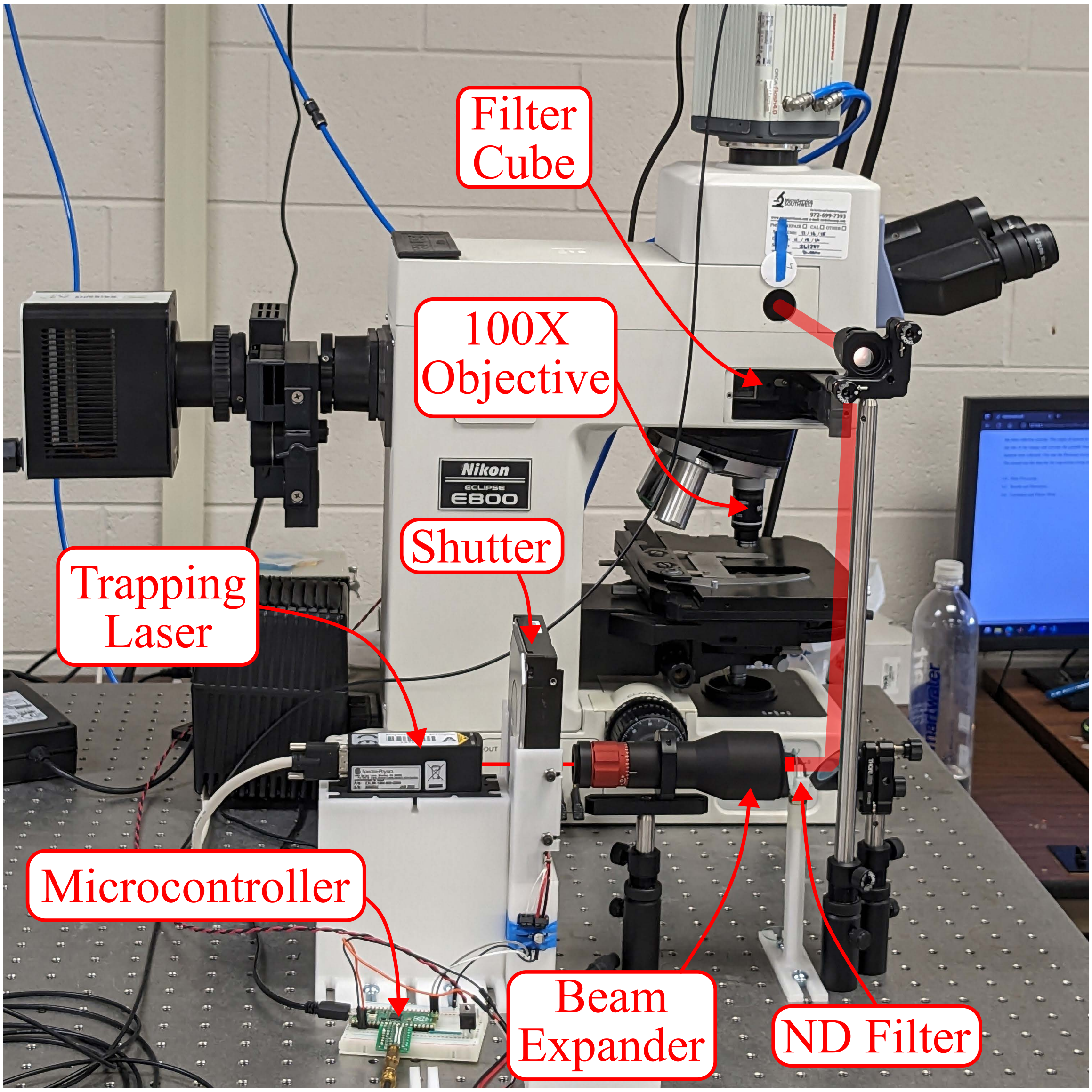


Figure 5.3: Side view of the experimental setup showing different components along with the laser path

the beam is larger than the objective back aperture, then the beam output from the objective will have an incomplete Gaussian profile. On the other hand, if the beam is smaller than the objective back aperture, then the high angle of divergence of the beam achievable, due to the high NA of the objective, will not be realized.

Samples were illuminated using a blue 3Watt LED light through a blue/green excitation-emission filter cube to achieve high contrast for particle tracking. However, the light from this LED was not dense enough to illuminate 500nm particles properly. This was tackled by illuminating the sample using a Cyan-colored (488nm wavelength) laser, GH04850B2G from Sharp, from the condenser side. Particle trajectories were captured with a high-speed digital camera, Hamamatsu Orca Flash 4. An infrared (IR) blocking filter was used to prevent the back-scattered laser light from reaching the camera.

The camera internally uses a Peltier device to actively cool the image sensor. A heat exchanger with a fan is present in the camera that dissipates the heat during normal operation. However, the vibrations from the fan were interfering with the PSD analysis. Thus, a water-cooling apparatus was used to cool the camera and prevent the onboard fan from running. The water cooling setup included a water pump with a small reservoir and a finned heat exchanger with a fan. Both of these components were located on a separate table from the optical table, RS4000 from Newport, on which the rest of the optical trapping setup was located.

To conduct the TRR experiments, it was important to have real-time control over the shutter, illumination system and camera recording. This syncing was achieved using a Raspberry Pi Pico microcontroller (μC). The program for the μC was designed to first close the shutter, then send a signal to the camera to trigger the recording and finally reopen the shutter. On top of controlling the timings for an experiment, the μC code was also designed to allow the user to control everything manually. The complete code of the μC is available in Appn. G. Frames recorded by the camera were sent to the host computer through a Camera Link Frame Grabber, FireBird 1XCLD-2PE8 from Active Silicon. The communication between the μC and the host computer was handled through a USB Serial interface. All the aspects of communication with the camera and the μC combined were managed by a singular software that was developed in-house, provided in Appn. H.

5.3 Experimentation Procedures

Green fluorescent polystyrene particles with diameters of 500nm, 1μm and 2μm were used for the experiments. The particles were first diluted in distilled water. It was desirable to reduce the particle density in the solution to improve the chances of trapping singular particles. The solution was filled into a cavity created by sandwiching multiple layers of 100μm thick double-sided tape between a glass slide and a cover slip as shown in Fig. 5.4. Precision coverslips with a thickness of 170μm, recommended by Nikon for the objective[66], were used to minimize the amount of spherical aberration[27].

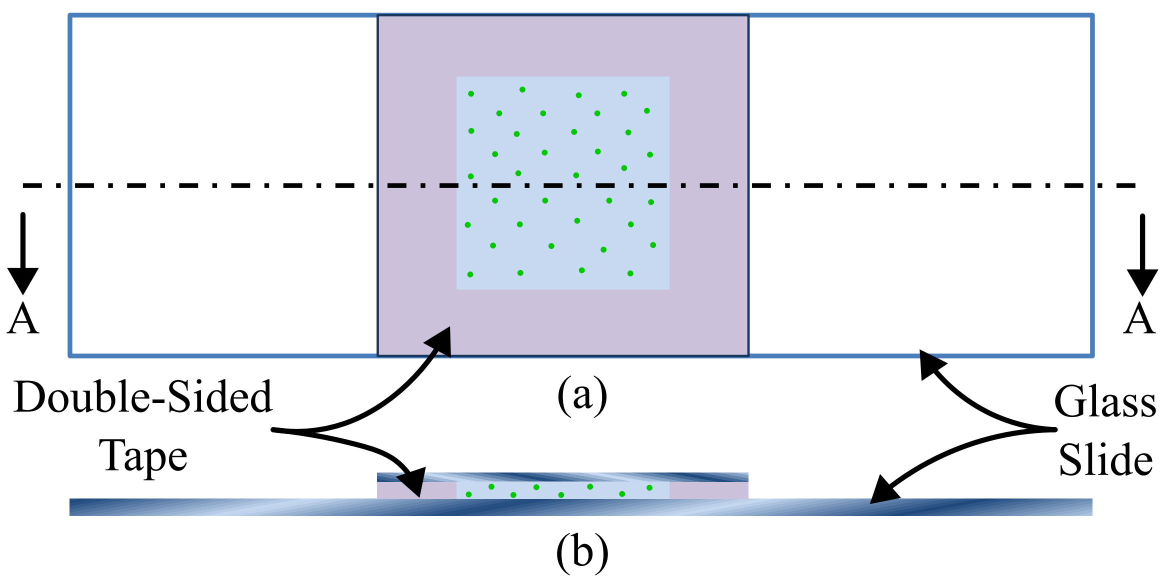


Figure 5.4: Sample created by filling the solution with particles into a cavity created by sandwiching double-sided tape between a glass slide and a cover slip. (a) Top view, (b) Cross-sectional view

The second important step during experiments was the alignment of the laser beam. The location and angle at which the beam enters the back aperture of the objective lens changes the trapping location and trap orientation[52, 62]. Thus, it was necessary to make sure that the beam entered at the center of the objective back aperture with an angle parallel to the longitudinal direction of the objective lens. A custom objective lens was developed to assist with the alignment of the laser beam entering the back aperture of the objective lens. This custom objective had acrylic plates at both ends with a cross-hair pattern engraved on them. If the resulting output of this custom objective showed only one cross-hair pattern then it meant that the beam was aligned with the longitudinal direction of the objective lens. The exploded rendering of this custom objective is shown in Fig. 5.5(a) and a sample of the output beam in an aligned state is shown in Fig. 5.5(b).

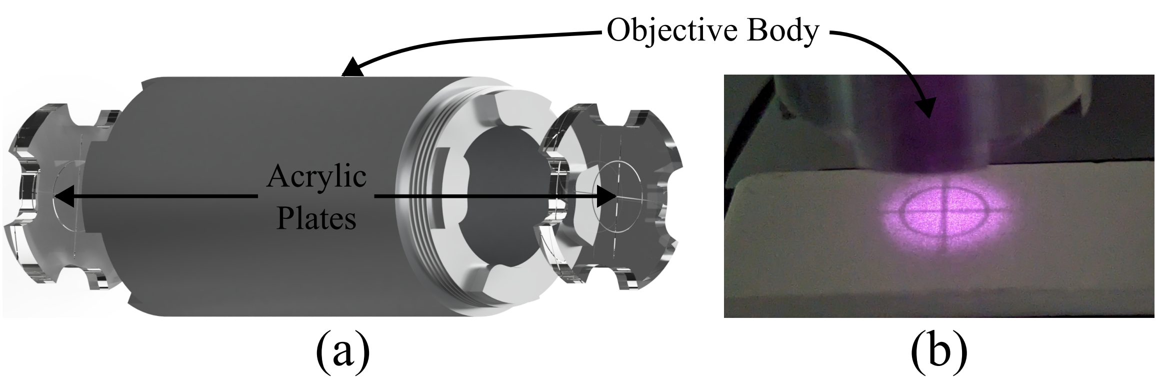


Figure 5.5: (a) Rendered image of the custom calibration objective, (b) Output of the custom objective when the laser is aligned well with the longitudinal direction of the objective

Another important aspect that determines the quality of the optical trap is the beam profile at the specimen plane. This was difficult to measure using a 100X oil immersion objective. Thus, a 20X air objective, from Nikon, with an NA of 0.75 was used along with a laser detection card. The surface of the detection card was brought to focus, and the emitted green light was imaged revealing the relative intensity profile of the beam. However, the detection card surface is uneven which results in hot and cold spots in the image. This was resolved by capturing multiple images at different locations of the detection card and averaging the relative intensities. The resulting beam profile is shown in Fig. 5.6 along with a Gaussian curve fit. Matlab's `lsqcurvefit` was used to fit the relative intensity values to a 2D Gaussian function,

$$I(x,y) = B + A \cdot \exp\left(-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)\right) \tag{5.1}$$

where, B is the bias, A is the amplitude, (x_0, y_0) represent the location of the peak intensity and (σ_x, σ_y) are the standard deviations. Note that the beam intensity profile is close to being Gaussian. The full width at half maximum (FWHM) and the beam diameter, in Fig. 5.6, are defined as 2.35482σ and 4σ respectively [99]. The curve fit parameters after optimization were found to be

$$\begin{aligned} B &= 0.0174 \quad x_0 = 502.8702\text{px} \quad \sigma_x = 132.1023\text{px} \\ A &= 0.9053 \quad y_0 = 491.4815\text{px} \quad \sigma_y = 135.4913\text{px} \end{aligned} \tag{5.2}$$

This step also helped in calculating the approximate location of the trap in the image which will later be helpful with trapping the particles and maximizing the camera capturing rate.

The experiments can begin once the laser is aligned and the beam profile is measured. To conduct these experiments, the inner surface of the coverslip was first brought to focus. This was done by finding some particles that were stuck to the surface and did not show any Brownian motion. Experimental observations showed that trapping particles farther away from the coverslip, greater than $\approx 80\mu$m, was difficult. This effect can be attributed to the increase in the aberration as the distance from the coverslip increased. Thus, locating the coverslip surface helped find free

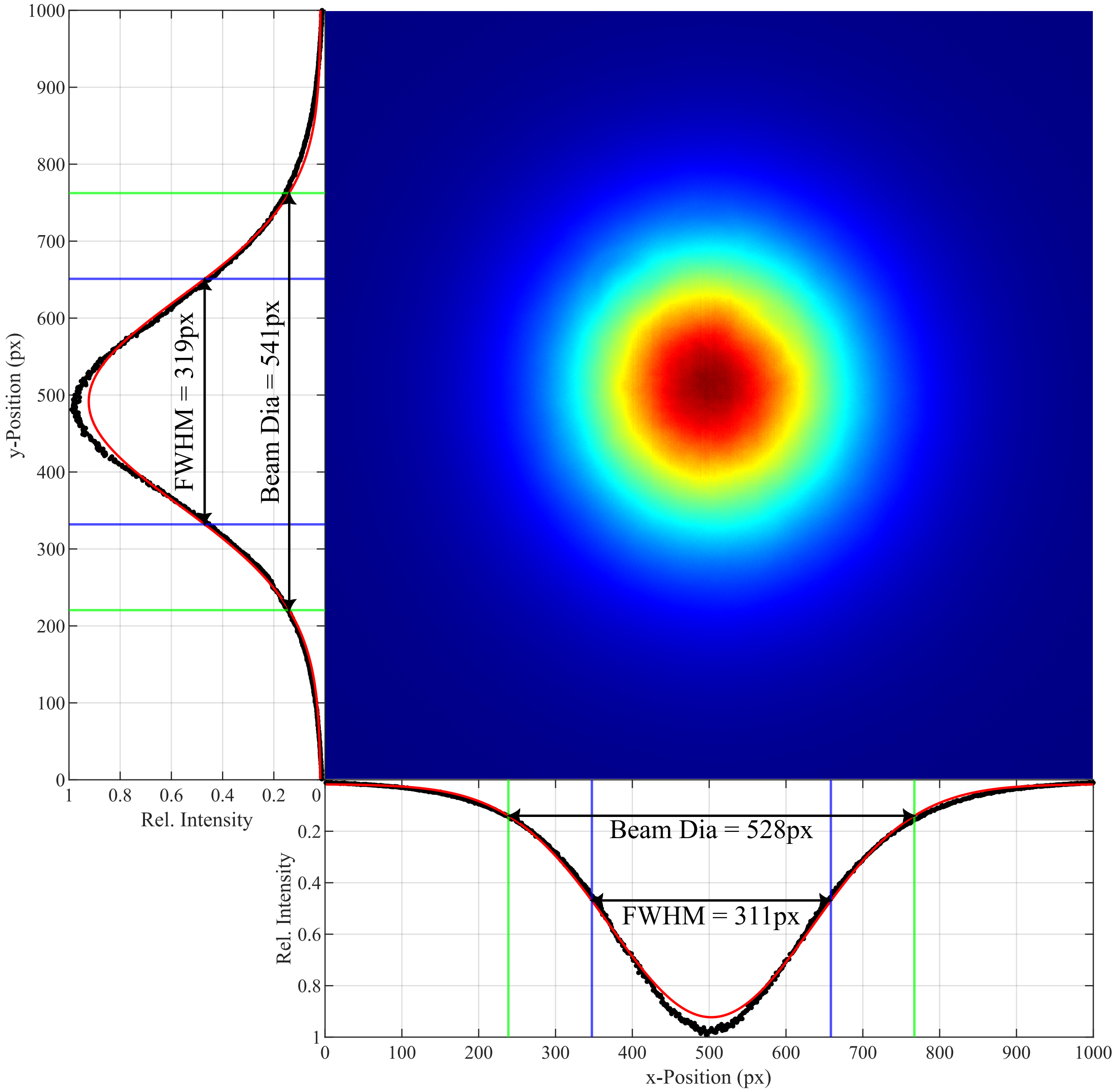


Figure 5.6: Laser beam profile at the sample plane in relative intensity units, captured using 20X objective and a laser detection card

particles that were close to it. Till this point, the trap is kept off by keeping the shutter closed.

A free particle is brought closer to the trap location in the image and the shutter is opened to trap it. The stage can be moved slowly to shift the trapped particle into a low particle density region ensuring that no other particle will get trapped during the data collection process. The region of interest in the image is changed to reduce the size of the images and achieve a high frame rate. Two different types of datasets were collected. One was the Brownian motion data of the trapped particle and the second was the data for the TRR experiment. The Brownian motion data of the trapped particle was collected for roughly 60 seconds while each TRR experiment was ≈ 1 second long with different shutter closing times for different particle sizes.

5.4 Data Processing

5.4.1 Data Compression

The data collected from experiments is first compressed before conducting further analysis. This step may not seem important for the research, however, it is very important for long-term storage since the amount of data generated is huge. The intensity value represented by each pixel of the image is 16 bits in depth, i.e. each pixel value can fall between 0 and 65535 and thus requires 2 bytes of storage. Considering the fact that the Brownian motion data for a 1μm particle is collected at $40\text{px} \times 64\text{px}$ image size and ≈ 5100 frames per second, one Brownian motion dataset will be $40\text{px} \times 64\text{px} \times 2\text{bytes/px} \times 60\text{sec} \times 5100\text{frames/sec} \approx 1.46\text{GBytes}$ in size.

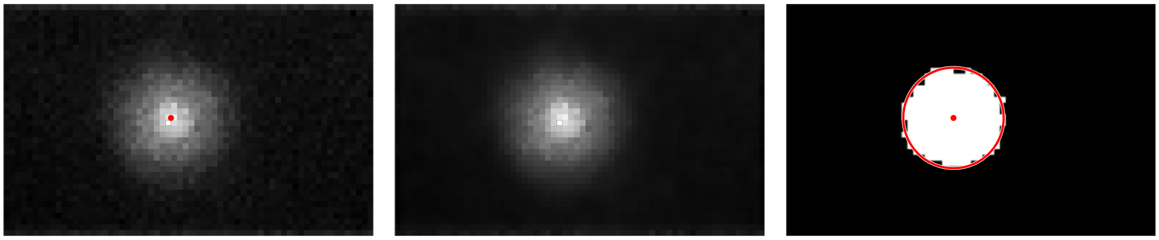
There exist multitudes of options for image and video compression; like converting the images to JPG or PNG format, or converting the video to MP4 or AVI format. However, these image/video formats perform lossy compression, i.e. some information is lost. The goal here was to attain the highest possible compression in lossless form. Two ways were encountered to achieve this, (1) Converting images into a video with Motion JPEG 2000 (MJ2) file format with lossless compression and (2) Accumulating all the images into a SuperFrame[37] image with JPEG 2000 (JP2) file format with lossless compression. The MATLAB codes for both approaches are available in Appn. I.

Out of the two approaches, the SuperFrame approach gave better results, however, it was a bit difficult to implement compared to the first approach. Regardless, the Superframe approach achieved a compression ratio close to 3 while the video compression approach achieved a compression ratio of close to 2.3. However, during the implementation of the Superframe approach, a bug in MATLAB was encountered where a high image aspect ratio would cause the software to crash. Thus, the video compression approach was used in this work which made the 1.46GBytes of Brownian motion data for the 1μm particle compress down to $\approx$ 0.63GBytes.

5.4.2 Centroid Detection

Many techniques[29, 77, 57] exist that can help extract the position of a particle from a gray-scale image. This work uses two techniques, (1) Image binarization and region detection and (2) Region matching using cross-correlation[29]. Although each technique used here performs well on its own, every image processing algorithm has its pitfalls. Thus, the reliability of the data can be assessed by comparing the results of the two approaches used here.

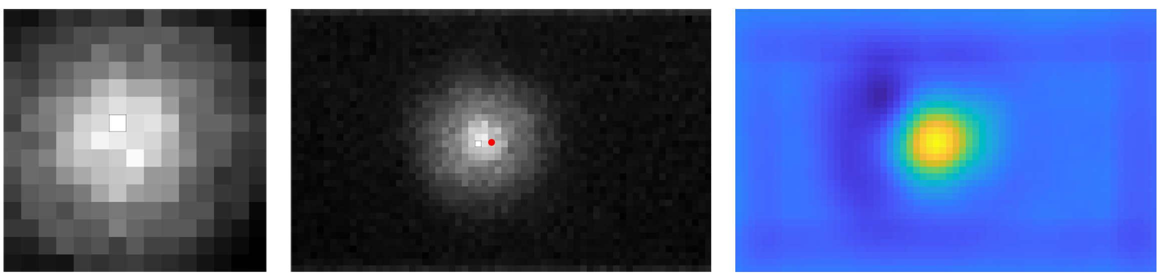
The first approach, referred to as the 'imBinarize' approach from here on out, takes advantage of the fact that there is only one bright object in the image, i.e. the fluorescent particle. Figure 5.7 shows the steps taken to calculate the centroid of the particle. The first step is to use `weiner2` filter from MATLAB to reduce the background noise. After this, the image can be binarized using `imbinarize`. All the connected regions in the binarized image can now be labeled using `bwlabel`. Finally, the `regionprops` function can be used to extract information regarding each detected



(a) (b) (c)

Figure 5.7: Visualization of image binarization and region detection (imBinarize) algorithm: (a) Image captured by the camera along with the computed particle centroid represented by a red dot (●), (b) Image filtered using `weiner2` function in MATLAB, (c) Region in the image that represents the particle along with the computed centroid (●) and the circle perimeter (—)

region. The resulting binary image, particle centroid and the circle representing the particle are visualized in Fig. 5.7(c).



(a) (b) (c)

Figure 5.8: Visualization of region matching using cross-correlation (kernelFit) approach: (a) Kernel Image extracted from one of the images in the recording, (b) Image taken by the camera along with the computed particle centroid represented by a red dot (●), (c) Cross-Correlation matrix plotted as an image showing that the highest cross-correlation is achieved when the kernel image overlaps the particle

The approach of region matching using cross-correlation, referred to as the 'kernelFit' approach from here on out, starts with first choosing a kernel image, i.e., an image that represents a particle, out of a big image captured by the camera. The camera image is stored in a matrix $I(x, y)$ with size $p \times q$ pixels and the kernel extracted from this image is stored in a matrix $K(i, j)$ with size $r \times s$ pixels. It is assumed that the center pixel of the kernel image, referred to as pixel $K(0,0)$, is the centroid of the particle. A cross-correlation matrix of the same size as that of the image can be computed using the following formula

$$C(x,y) = \sum_{i=\frac{1-r}{2}}^{\frac{r-1}{2}} \sum_{j=\frac{1-s}{2}}^{\frac{s-1}{2}} (I(x+i,y+j) - I_{mean})(K(i,j) - K_{mean}) \tag{5.3}$$

where, I_{mean} and K_{mean} are the mean values of the complete image and kernel matrices respectively, and the pixel values $I(x+i, y+j)$ that are not within the bounds of matrix I are assumed to equal zero. A threshold value T can be chosen to extract the cross-correlation peak. The particle centroid can be computed by

$$x_c = \frac{\sum x \cdot (C(x,y) - T)}{\sum (C(x,y) - T)} \qquad y_c = \frac{\sum y \cdot (C(x,y) - T)}{\sum (C(x,y) - T)} \tag{5.4}$$

Figure 5.8 shows this process. Note that the particle coordinates found here, x_c and y_c, are in Pixel units.

Note that the detected centroids from both approaches are different. This is mainly because the kernelFit approach finds the center of the kernel image in the camera image instead of finding the particle centroid. Thus, the results are reliable as long as the difference in centroid from both approaches is constant. Figure 5.9 shows the centroid difference between the two approaches for a 70sec Brownian motion dataset of a 2μm diameter particle. Note that the standard deviation in the difference of the centroid values is ≈ 0.085px for both directions.

5.4.3 Trap stiffness estimation

Note that the coordinates of the particle centroid calculated in the last subsection are in pixel units. It is important to have these coordinates in physical units. The physical size of each pixel on the Hamamatsu camera is 6.5μm $\times$ 6.5μm. A conversion factor can be computed based on the objective zoom value that will allow

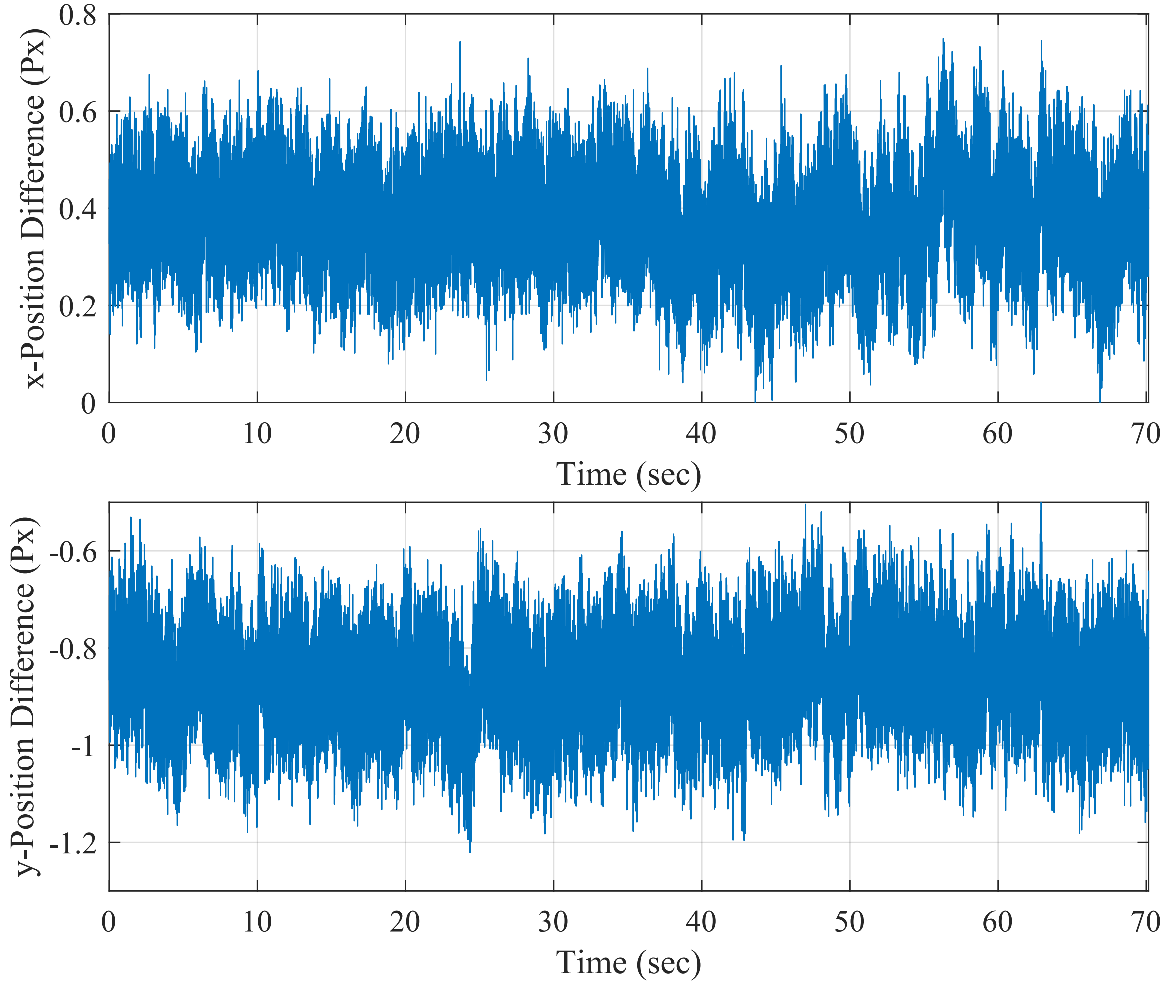


Figure 5.9: Difference in the detected particle centroid from the two centroid detection methods for a Brownian motion dataset of a 2μm diameter particle

the conversion of pixel coordinates to physical units. The conversion factor for the 100X objective used for the experiments will be.

$$c = \frac{\text{Camera Pixel Size}}{\text{Zoom of the Objective}} = \frac{6.5\mu\text{m/px}}{100} = 0.065\mu\text{m/px} \tag{5.5}$$

Many active and passive trap stiffness measurement techniques can be found in the literature[79, 44]. Each method has its advantages and shortcomings. This work utilizes a passive stiffness measurement technique, namely the Power Spectral Density (PSD) analysis discussed in Sec. 4.3. After converting the particle centroid values to physical units, the Fourier transform of the Brownian motion data is calculated, using `fft` function in MATLAB, to further calculate the PSD using Eqn. (4.7). The noisy raw PSD profile is then filtered by dividing the data into several bins and averaging the data of each bin. Finally, the theoretical equation of the PSD profile, Eqn. (4.12), is fitted to the filtered PSD profile using `lsqcurvefit` in MATLAB. The estimated value of the corner frequency, f_c, is used to calculate an estimate of the trap stiffness in both the x and y directions. The PSD curve fits for particles with different diameters and the corresponding trap stiffness values are presented in the next section.

5.5 Results and Discussion

The following subsections discuss the experimental PSD profiles and the curve fit of Eqn. (4.12) to the PSD profile along with the results of TRR experiments for the 2μm, 1μm and 500nm diameter polystyrene particles.

5.5.1 Data for 2μm diameter particle

Figures 5.10 and 5.11 show the experimental PSD profiles and Eqn. (4.12) fitted to these profiles for a 2μm diameter polystyrene particle. Specifically, Fig. 5.10 presents the PSD profile calculated from the particle position data that was extracted using the imBinarize approach. On the other hand, Fig. 5.11 presents the PSD profile calculated from the particle position data that was extracted using the kernelFit approach. The statistical values from five different Brownian motion datasets for the imBinarize approach are,

$$\begin{aligned} \bar{k}_x &= 75.379823\text{pN}/\mu\text{m} \quad & \bar{k}_y &= 78.893746\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 0.766195\text{pN}^2/\mu\text{m}^2 \quad & \sigma^2_{k_y} &= 0.731596\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.6}$$

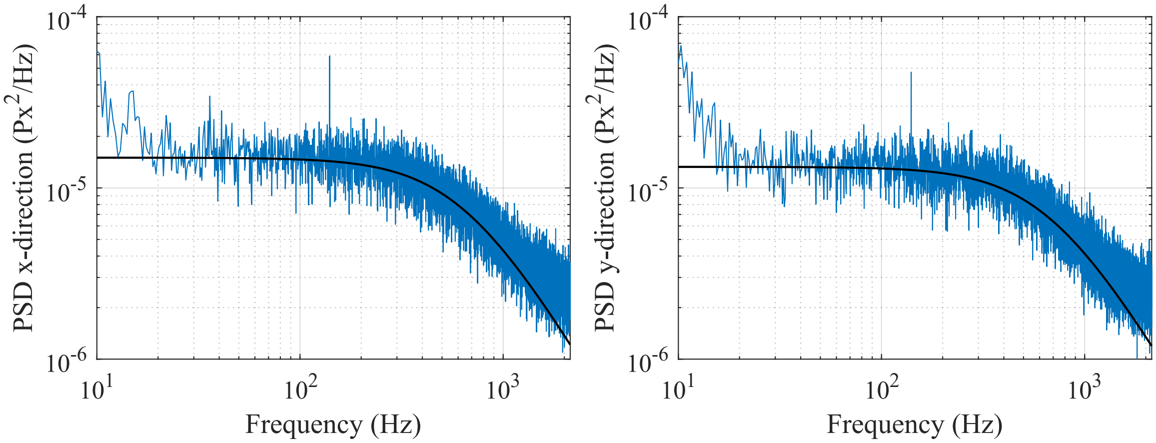


Figure 5.10: PSD profile for a 2μm diameter particle centroid computed using the imBinarize approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

where, $\bar{k}_x$ and $\bar{k}_y$ represent the average stiffnesses in x and y directions respectively, and $\sigma^2_{k_x}$ and $\sigma^2_{k_y}$ represent the corresponding variances. Similar values for the kernelFit approach are,

$$\begin{aligned} \bar{k}_x &= 73.451003\text{pN}/\mu\text{m} \quad & \bar{k}_y &= 76.546744\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 0.732422\text{pN}^2/\mu\text{m}^2 \quad & \sigma^2_{k_y} &= 0.255802\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.7}$$

Note that the stiffness values are quite similar between the two directions as well as between the two approaches. This means that the laser beam at the sample plane is close to symmetric and the stiffness values are reliable. The PSD profiles and curve fits for the other four Brownian motion datasets are available in Appn. B.1.

Figure 5.12 presents the data from the TRR experiment. These graphs present data from both the imBinarize and kernelFit approaches. It should be noted that the difference in the trajectory computed through both approaches has a constant difference as discussed in Subsection 5.4.2. For these experiments, the trap was turned off for 500ms so that the particle could wander off from the trap location and be retrapped. Note that the particle shows a pure Brownian motion for the

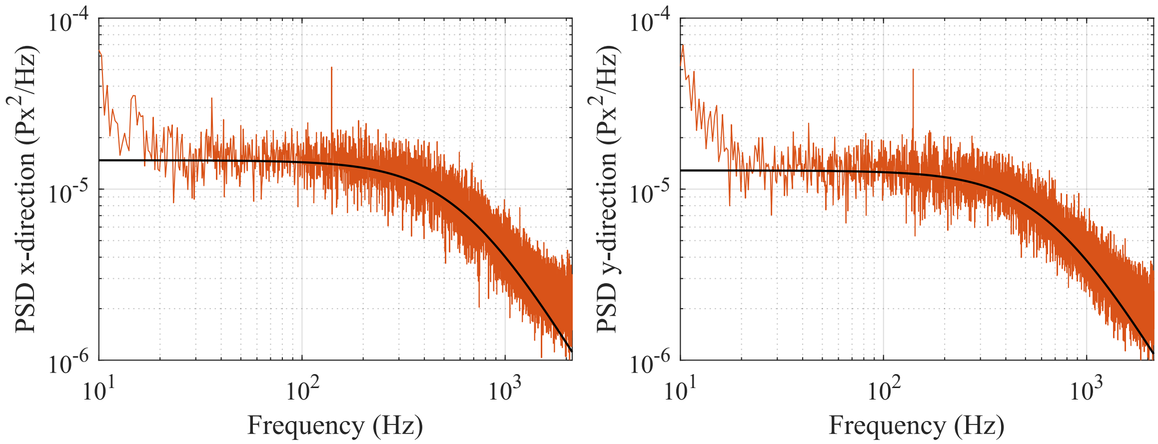


Figure 5.11: PSD profile for a 2μm diameter particle centroid computed using the kernelFit approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

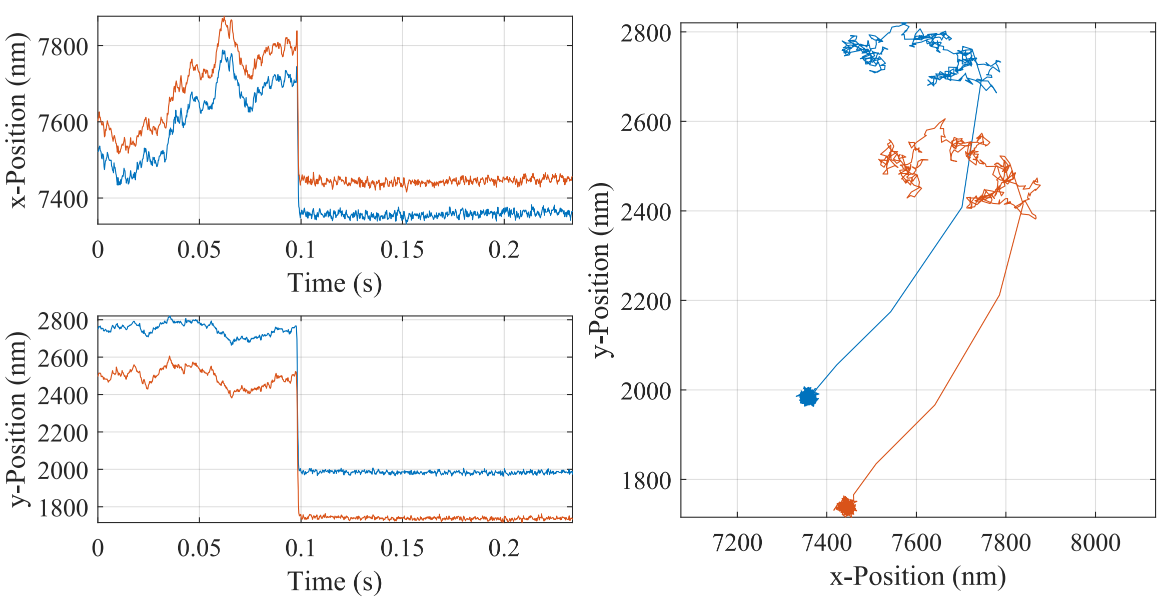


Figure 5.12: TRR data for a 2μm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches, the trap was turned off for 500ms in this experiment

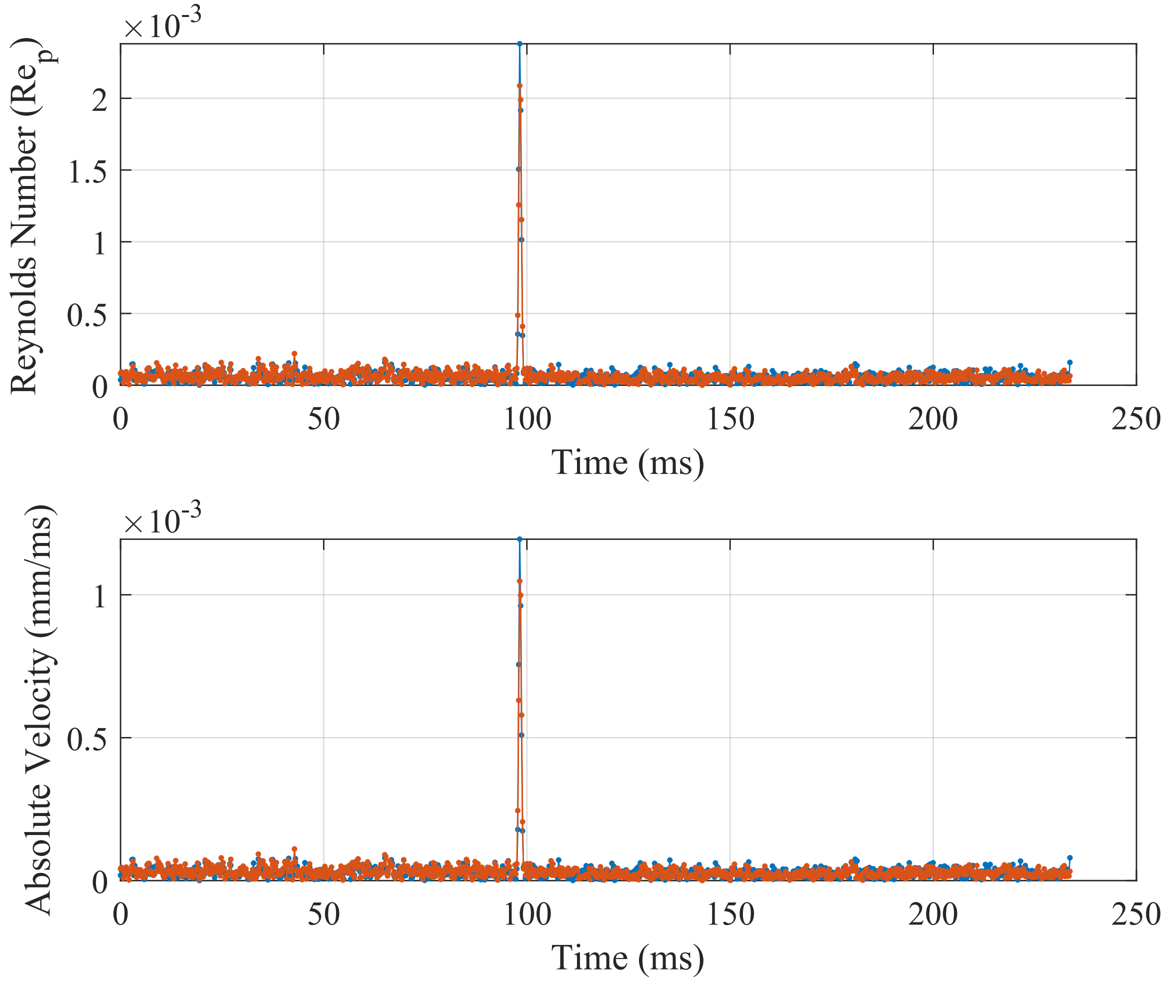


Figure 5.13: Particle Reynolds number and absolute velocity for a 2μm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches

first 100ms before it gets trapped. Also note that there is no underdamped behavior observed here similar to the previous dataset for the 1950nm diameter particle. Four more datasets of TRR experiments are available in Appn. C.

Figure 5.13 presents the particle Reynolds number ($\mathrm{Re_p}$), calculated using Eqn. (4.5), for the particle trajectory in the TRR experiment, Fig. 5.12. The particle velocity is calculated by numerically differentiating the particle position values in each direction. Note that the $\mathrm{Re_p}$ for the particle here is smaller than or equal to 2.5×10^{-3}. This is higher than the one observed in the previous dataset of 1950nm diameter particle, see Fig. 4.4. However, the difference is not too big to show any significant difference between the inertia forces and viscous forces.

5.5.2 Data for 1μm diameter particle

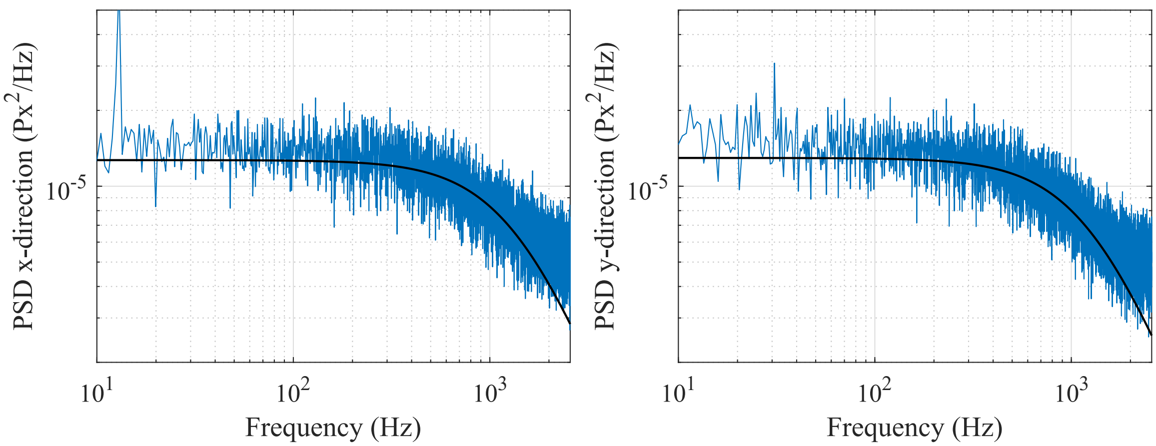


Figure 5.14: PSD profile for a 1μm diameter particle centroid computed using the imBinarize approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

The experimental PSD profile and the corresponding curve fit for 1μm diameter polystyrene particle are presented in Figs. 5.14 and 5.15. The statistical values from nine different Brownian motion datasets for the imBinarize approach are,

$$\begin{aligned} \bar{k}_x &= 81.148448\text{pN}/\mu\text{m} & \bar{k}_y &= 75.956058\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 18.859406\text{pN}^2/\mu\text{m}^2 & \sigma^2_{k_y} &= 18.762983\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.8}$$

Similar values for the kernelFit approach are,

$$\begin{aligned} \bar{k}_x &= 69.425459\text{pN}/\mu\text{m} & \bar{k}_y &= 64.393509\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 7.976650\text{pN}^2/\mu\text{m}^2 & \sigma^2_{k_y} &= 7.705721\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.9}$$

The PSD profiles and curve fits for the other eight Brownian motion datasets are available in Appn. B.2. It should be noted that even though the difference in the

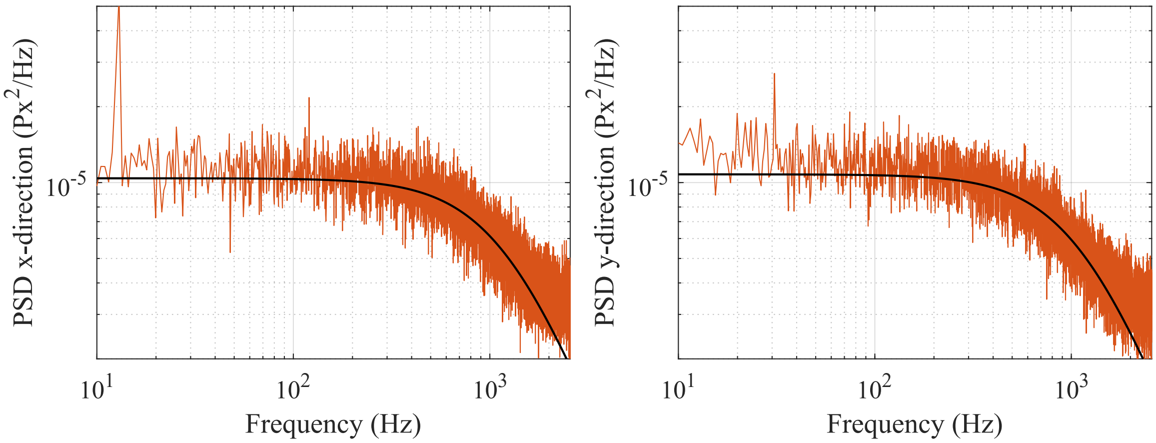


Figure 5.15: PSD profile for a 1μm diameter particle centroid computed using the kernelFit approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

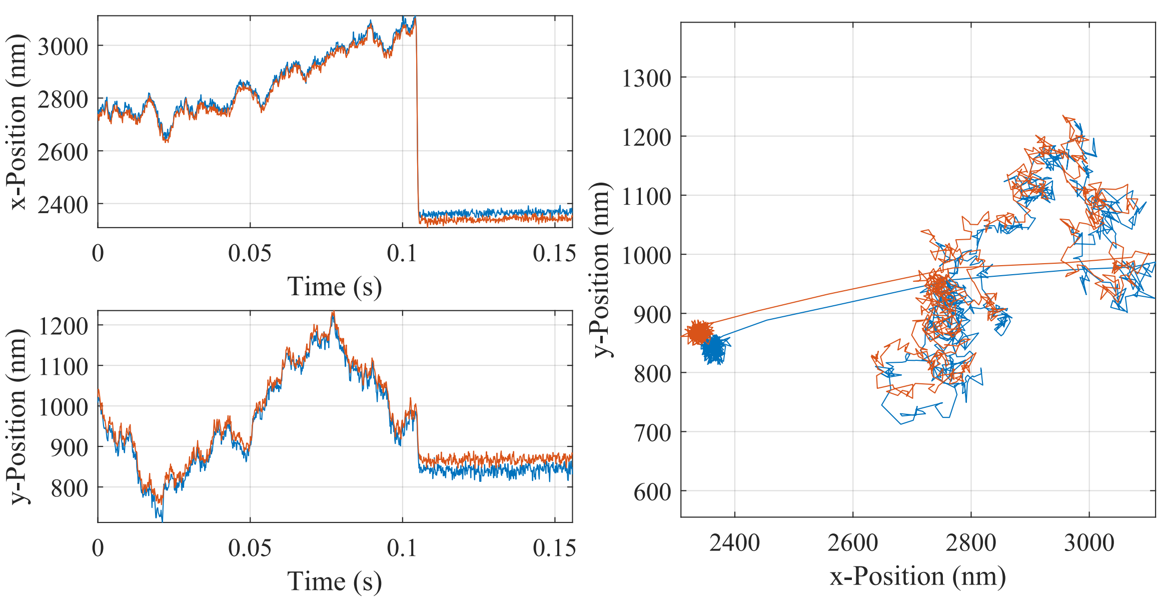


Figure 5.16: TRR data for a 1μm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches, the trap was turned off for 100ms in this experiment

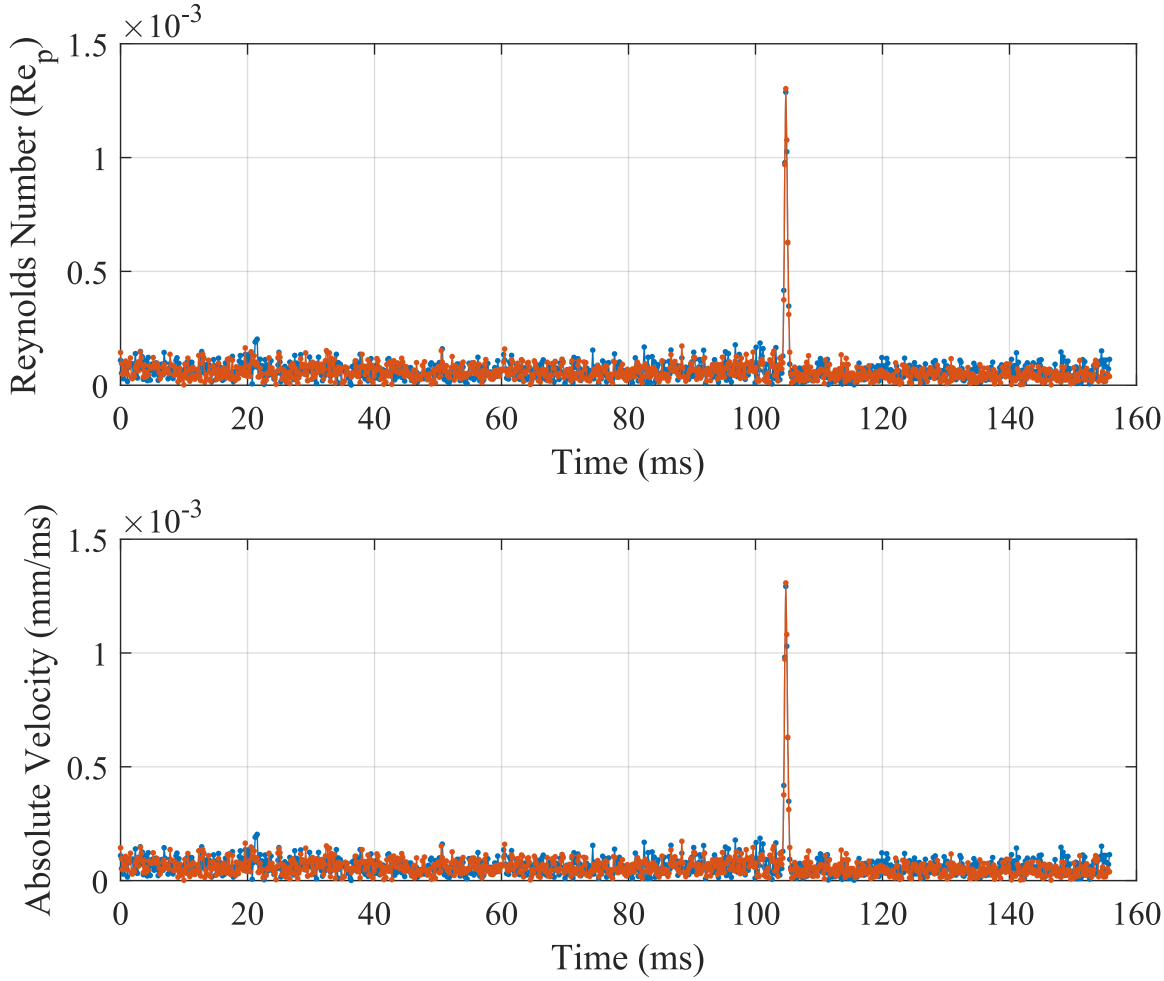


Figure 5.17: Particle Reynolds number and absolute velocity for a 1μm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches

stiffness values between the two directions and the approaches is small, the variances are much larger than the ones for the 2μm particle. Thus, the accuracy of the stiffness value might be lower. Furthermore, note that the magnitude of the stiffness values found here is similar to the ones for the 2μm particle. This is because the laser power used to collect these datasets was lower than the one used for the 2μm particle.

There has to be a balance between the framerate used to capture the video and the trap stiffness. As the framerate increases, the light available from the particle to be captured by the camera reduces, thus reducing the signal-to-noise ratio. As a result, it is difficult to achieve frame rates higher than $\approx$ 5000FPS with the current setup even though the camera is more capable than this. This puts a limit on the observable frequency range of the PSD profile as per the Nyquist theorem. Thus, the stiffness/power has to be low enough so that the corner frequency can fall in the observable frequency range and can be estimated accurately. The main goal of the PSD analysis here is to check the evenness of the stiffness values in both directions, which is achieved here.

Figure 5.16 presents a dataset from the TRR experiments. For these experiments, the trap was turned off for 100ms. Note that this time period was enough for the particle to drift $\approx$ 700nm away from the trapping location. The laser power used for these experiments was the same as the one used for the 2μm particle. This was desirable because the higher stiffness value, achieved using the higher laser power, would increase the chances of causing the underdamped behavior as per Eqn. (4.3). However, there is no underdamped behavior observed here, contrary to a small amount observed in the old dataset of the 990nm diameter particle. Thirty-seven more datasets of TRR experiments are available in Appn. D.

Figure 5.17 presents the particle Reynolds number ($\mathrm{Re_p}$). Note that the $\mathrm{Re_p}$ here is smaller than or equal to 1.4×10^{-3}. Similar to the 2μm particle, the $\mathrm{Re_p}$ is higher than the one observed in the previous dataset of 990nm diameter particle, see Fig. 4.5. This should mean that the effect of inertia forces should be higher than the viscous forces here. However, the underdamped behavior is nonexistent here while the previous dataset showed a small amount of it.

5.5.3 Data for 500nm diameter particle

The particles were illuminated using only the blue LED in all the experiments so far. However, the available fluorescent light from the 500nm diameter particle was

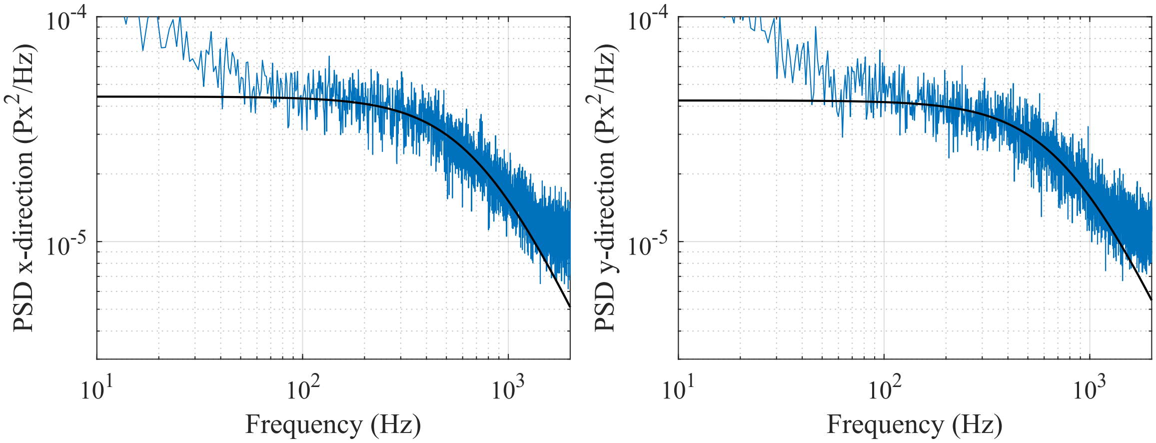


Figure 5.18: PSD profile for a 500nm diameter particle centroid computed using the imBinarize approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

not enough, while using only the LED, to achieve high framerates. The addition of the Cyan colored laser resolved this problem and achieved the camera framerate of 4000FPS.

The trapping laser power was lowered even further for the Brownian motion experiments of the 500nm diameter particle. The experimental PSD profile and the corresponding curve fit are presented in Figs. 5.18 and 5.19. The statistical values from five different Brownian motion datasets for the imBinarize approach are,

$$\begin{aligned} \bar{k}_x &= 21.388566\text{pN}/\mu\text{m} & \bar{k}_y &= 22.447706\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 0.517168\text{pN}^2/\mu\text{m}^2 & \sigma^2_{k_y} &= 0.640007\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.10}$$

Similar values for the kernelFit approach are,

$$\begin{aligned} \bar{k}_x &= 18.438312\text{pN}/\mu\text{m} & \bar{k}_y &= 19.416466\text{pN}/\mu\text{m} \\ \sigma^2_{k_x} &= 0.460043\text{pN}^2/\mu\text{m}^2 & \sigma^2_{k_y} &= 0.409082\text{pN}^2/\mu\text{m}^2 \end{aligned} \tag{5.11}$$

The PSD profiles and curve fits for the other four Brownian motion datasets are available in Appn. B.3. Note that the difference in the stiffness values between the two directions and the approaches is small along with the low variance values

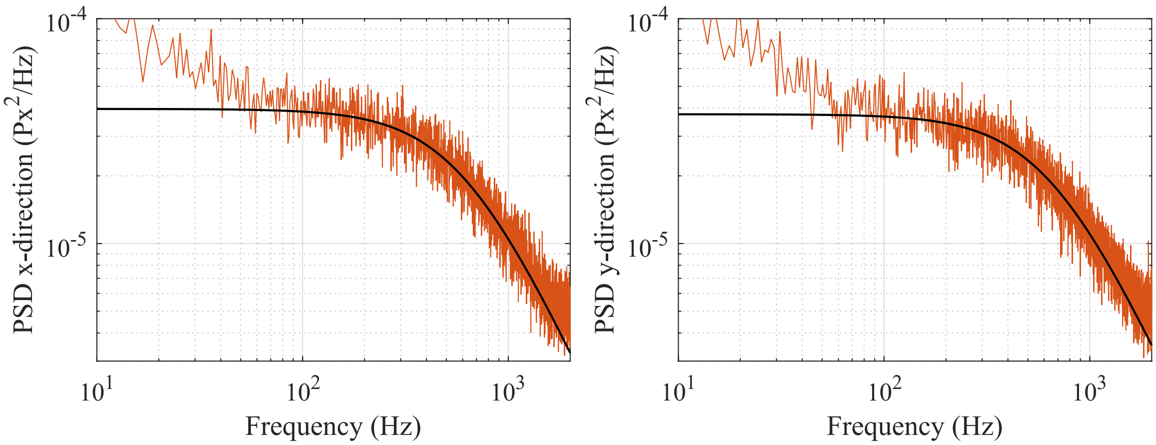


Figure 5.19: PSD profile for a 500nm diameter particle centroid computed using the kernelFit approach (—) along with the curve fit of Eqn. (4.12) to the PSD profile (—)

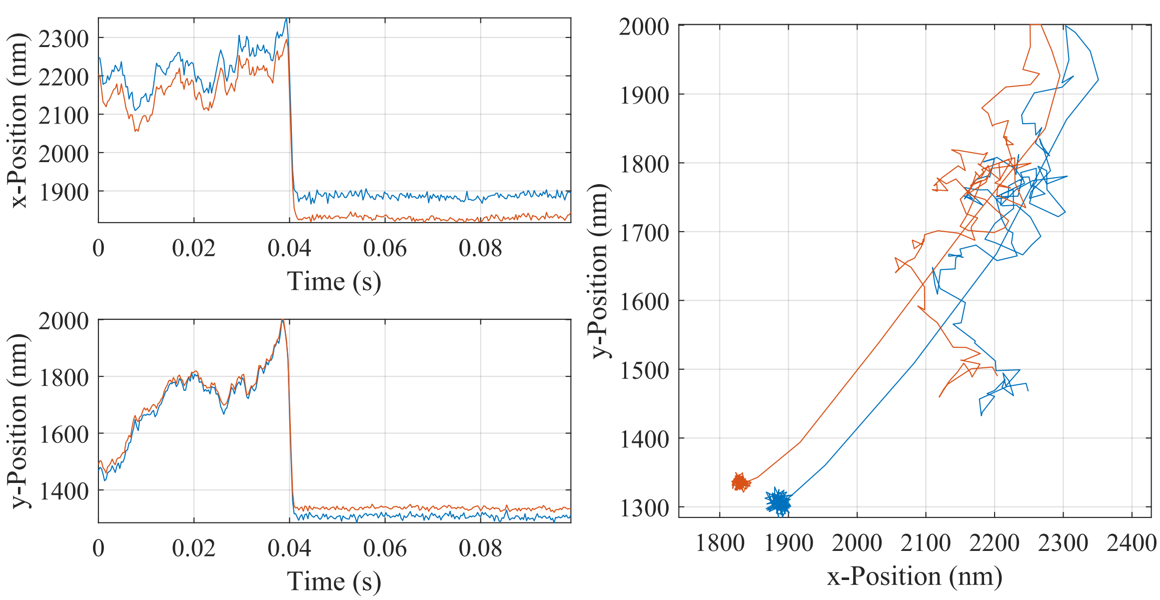


Figure 5.20: TRR data for a 500nm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches, the trap was turned off for 30ms in this experiment

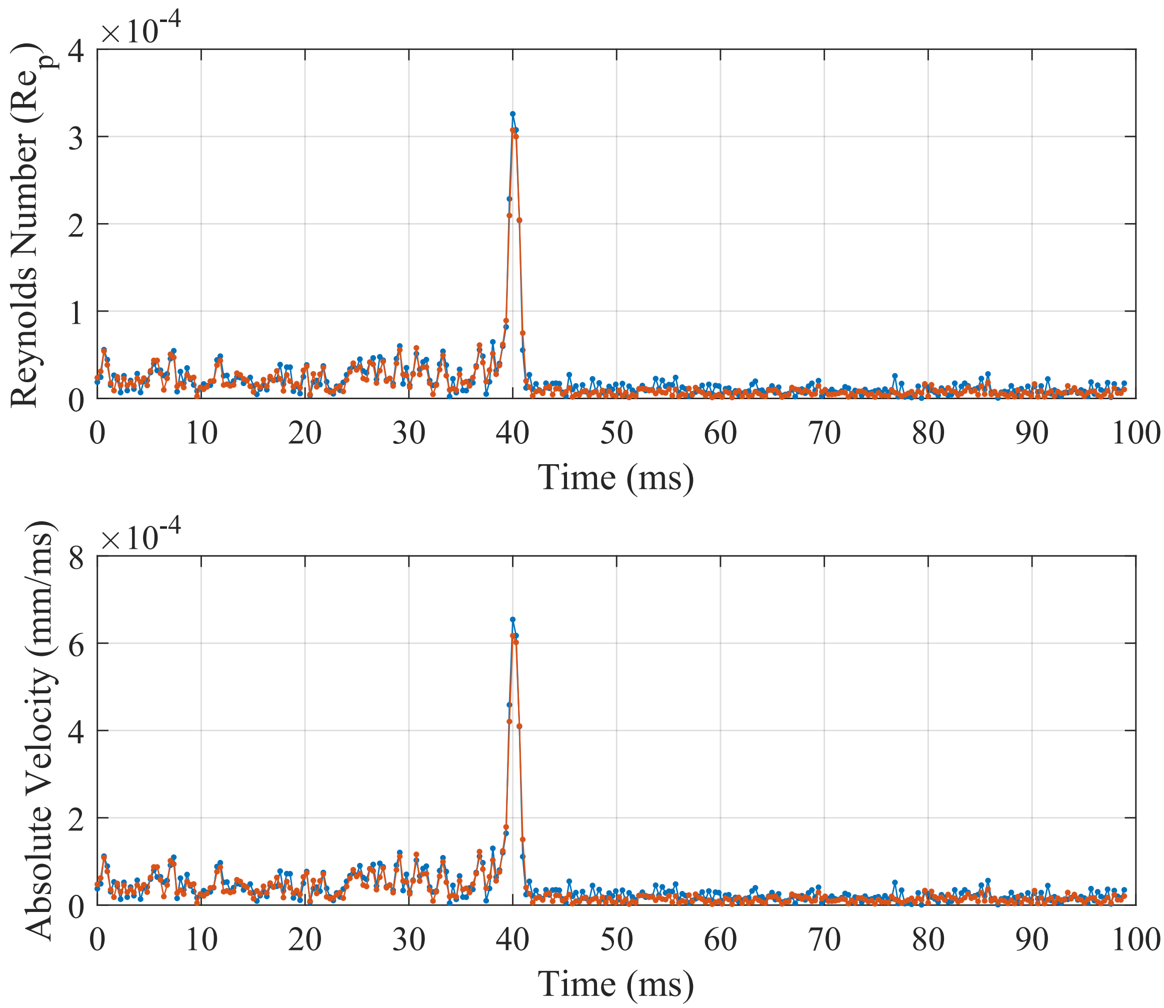


Figure 5.21: Particle Reynolds number and absolute velocity for a 500nm diameter particle computed using both the imBinarize (—) and the kernelFit (—) approaches

compared to the 1μm dataset. This was achieved because the use of the Cyan laser improved the signal-to-noise ratio significantly. However, the particle would lose its ability to fluoresce if it was illuminated using the laser for more than 60 seconds. Thus, the Brownian motion datasets collected here were only 30 seconds long. Furthermore, note that no peak is visible in the PSD profile near the corner frequency as what was suggested by the simulated PSD from the scaled model for 500nm diameter particle in Section 4.5.

Data for the TRR experiment is presented in Fig. 5.20. The trap was turned off for 30ms in this case. Note that the particle is $\approx$ 800nm away from the trapping location when the trap is turned on at 0.04s. The laser power used here was kept the same as the one used for the 2μm particle to increase the chances of causing the underdamped behavior. However, no underdamped behavior was observed here, compared to what was observed in the old dataset. Thirty-nine more datasets of TRR experiments are available in Appn. E.

the particle Reynolds number ($\mathrm{Re_p}$) is presented in Fig. 5.21. Note that the $\mathrm{Re_p}$ here is smaller than or equal to 3.5×10^{-4}. This value is extremely close to the one observed in the previous dataset of 500nm diameter particle, $\mathrm{Re_p} \leq 3 \times 10^{-4}$, Fig. 4.6. Although this value of $\mathrm{Re_p}$ was enough in the old dataset to show an underdamped motion, no underdamped motion was observed in this case.

5.6 Conclusion and Future Work

A new experimental setup for trapping micro/nano particles reliably and repeatedly is presented here with the primary goal of characterizing the suspected underdamped behavior that was observed in the previous dataset. This new setup is shown to have a measurably good quality of the laser beam intensity profile and the correct alignment with the objective, which should have minimized the light aberrations. Different methods and techniques of data compression, particle centroid detection and trap stiffness estimation were also presented here.

New experimental datasets for 2μm, 1μm and 500nm diameter particles were collected for both the Brownian motion experiments and the TRR experiments. No significant underdamped behavior was observed in the 40 TRR trajectories for the 500nm diameter particle presented here. Furthermore, it was discussed in Section 4.5 that a distinct peak should be visible in the PSD profile of the 500nm diameter particle's position if the particle was supposed to show the underdamped motion. However, no such peak was observed in the new data presented here. Currently, the findings of the new dataset contradict the old experimental data and the hypothesis that was developed based on the previous observations.

One conclusion that can be drawn here is that if the laser beam profile has a good quality factor (M^2) and a correct alignment is achieved then the underdamped motion does not occur. Thus, the possible reasons for the underdamped motion in the old datasets could either be a poor beam quality or some type of fluid force effect that was not captured in the experiments presented here. Unfortunately, no data exists for the beam quality of the previous experiment.

The best way forward is to complete the experiments of transporting the particles through the "L" shaped microchannel as discussed in Section 4.6. If the data from these experiments deviate from the Stokes drag model then the fluid forces action on the particle can be analyzed in further detail. The current trapping setup can also be improved to reproduce the conditions encountered in the microchannel experiments.

Regardless of the steps that are taken in the future, the current optical trapping setup is already being pushed to its limits. The primary challenge with the current setup is the insufficient amount of fluorescent light emitting from the 500nm diameter particle. This can be improved by using an objective lens with higher NA. However, the problem may arise again as the particle size is reduced further. Furthermore, the particle will barely be visible through a camera as its size reduces. Thus, for the experiments with smaller particles, either a Quadrant Photodiode or a Lateral Effect Position Sensor can be used, both of which provide better accuracy and higher sampling rate.

APPENDIX A

Equations of Motion for Double Pendulum System

The equations of motion for the double pendulum example are computed using Kane's method. Because the final EOMs are too large, this section does not derive complete EOMs. Rather it provides important equations in a format discussed in [15], which when evaluated result in the EOMs for the double pendulum.

A.1 Equations of Motion

Rigid Bodies: There are two rigid bodies, the two cylindrical rods of double pendulum.

Inertial Reference Frame and Point: The inertial reference point is N, and N $= \left(\widehat{\mathbf{N}}_1, \widehat{\mathbf{N}}_2, \widehat{\mathbf{N}}_3\right)$ is the inertial reference frame shown in Fig. 2.2.

Other Points and Frames: Point A and B are body-attached points, and they are also the center of gravity of bodies A and B respectively. Point C represents the spherical joint between two rods. A $= \left(\widehat{\mathbf{A}}_1, \widehat{\mathbf{A}}_2, \widehat{\mathbf{A}}_3\right)$ and B $= \left(\widehat{\mathbf{B}}_1, \widehat{\mathbf{B}}_2, \widehat{\mathbf{B}}_3\right)$ are body-attached frames shown in Fig. 2.2.

Location Descriptions: The location descriptions provide a formal way of specifying the position of every point that comprises the rigid bodies in the system.

$$L_A = \left\{\mathbf{P}_{NA}, {}^N_A R\right\} + geom. \quad L_B = \left\{\mathbf{P}_{CB}, {}^A_B R\right\} + geom. \tag{A.1}$$

$$\mathbf{P}_{NA} = \mathbf{P}_{AC} = l_A \widehat{\mathbf{A}}_1 \quad \mathbf{P}_{NC} = 2 l_A \widehat{\mathbf{A}}_1 \quad \mathbf{P}_{CB} = l_B \widehat{\mathbf{B}}_1 \tag{A.2}$$

$${}^N_A R = \frac{1}{e_{A_0}^2 + e_{A_1}^2 + e_{A_2}^2 + e_{A_3}^2} \begin{bmatrix} e_{A_0}^2 + e_{A_1}^2 - e_{A_2}^2 - e_{A_3}^2 & 2e_{A_1}e_{A_2} - 2e_{A_0}e_{A_3} & 2e_{A_0}e_{A_2} + 2e_{A_1}e_{A_3} \\ 2e_{A_1}e_{A_2} + 2e_{A_0}e_{A_3} & e_{A_0}^2 - e_{A_1}^2 + e_{A_2}^2 - e_{A_3}^2 & 2e_{A_2}e_{A_3} - 2e_{A_0}e_{A_1} \\ 2e_{A_1}e_{A_3} - 2e_{A_0}e_{A_2} & 2e_{A_2}e_{A_3} + 2e_{A_0}e_{A_1} & e_{A_0}^2 - e_{A_1}^2 - e_{A_2}^2 + e_{A_3}^2 \end{bmatrix} \tag{A.3}$$

$$
{}^{A}_{B}R \;=\; \frac{1}{e_{B_0}^2+e_{B_1}^2+e_{B_2}^2+e_{B_3}^2}
\begin{bmatrix}
e_{B_0}^2+e_{B_1}^2-e_{B_2}^2-e_{B_3}^2 & 2e_{B_1}e_{B_2}-2e_{B_0}e_{B_3} & 2e_{B_0}e_{B_2}+2e_{B_1}e_{B_3} \\
2e_{B_1}e_{B_2}+2e_{B_0}e_{B_3} & e_{B_0}^2-e_{B_1}^2+e_{B_2}^2-e_{B_3}^2 & 2e_{B_2}e_{B_3}-2e_{B_0}e_{B_1} \\
2e_{B_1}e_{B_3}-2e_{B_0}e_{B_2} & 2e_{B_2}e_{B_3}+2e_{B_0}e_{B_1} & e_{B_0}^2-e_{B_1}^2-e_{B_2}^2+e_{B_3}^2
\end{bmatrix} \qquad \text{(A.4)}
$$

Coordinates: Eight coordinates appear in the location descriptions. Where, $e_{A_{0-3}}$ and $e_{B_{0-3}}$ are Euler parameters representing body A's and body B's orientation respectively. The vector of generalized coordinates, $\mathbf{q}$ is defined as:

$$
\mathbf{q} \;=\; [\mathbf{e}_A^T \;\; \mathbf{e}_B^T]^T \;=\; [e_{A_0}\; e_{A_1}\; e_{A_2}\; e_{A_3}\; e_{B_0}\; e_{B_1}\; e_{B_2}\; e_{B_3}]^T
$$

Constraints: The normalization constraint associated with both the Euler parameter sets are given as

$$
q_1^2+q_2^2+q_3^2+q_4^2 \;=\; 1 \qquad , \qquad q_5^2+q_6^2+q_7^2+q_7^2 \;=\; 1
$$

Degrees of Freedom (DOFs):

8 coordinates - 2 constraint = 6 DOFs

Velocity: The angular velocity of body A and body B is

$$
\begin{aligned}
{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} &= 2\left(u_1\widehat{\mathbf{A}}_1 + u_2\widehat{\mathbf{A}}_2 + u_3\widehat{\mathbf{A}}_3\right) \\
&= 2(q_1\dot{q}_2 - q_2\dot{q}_1 - q_3\dot{q}_4 + q_4\dot{q}_3)\widehat{\mathbf{A}}_1 \\
&\quad + 2(q_1\dot{q}_3 + q_2\dot{q}_4 - q_3\dot{q}_1 - q_4\dot{q}_2)\widehat{\mathbf{A}}_2 \\
&\quad + 2(q_1\dot{q}_4 - q_2\dot{q}_3 + q_3\dot{q}_2 - q_4\dot{q}_1)\widehat{\mathbf{A}}_3
\end{aligned} \tag{A.5}
$$

$$
\begin{aligned}
{}^{\mathrm{A}}\boldsymbol{\omega}^{\mathrm{B}} &= 2\left(u_5\widehat{\mathbf{B}}_1 + u_6\widehat{\mathbf{B}}_2 + u_7\widehat{\mathbf{B}}_3\right) \\
&= 2(q_5\dot{q}_6 - q_6\dot{q}_5 - q_7\dot{q}_8 + q_8\dot{q}_7)\widehat{\mathbf{B}}_1 \\
&\quad + 2(q_5\dot{q}_7 + q_6\dot{q}_8 - q_7\dot{q}_5 - q_8\dot{q}_6)\widehat{\mathbf{B}}_2 \\
&\quad + 2(q_5\dot{q}_8 - q_6\dot{q}_7 + q_7\dot{q}_6 - q_8\dot{q}_5)\widehat{\mathbf{B}}_3
\end{aligned} \tag{A.6}
$$

$$
{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{B}} = {}^{A}_{B}R^{T}\,{}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} + {}^{\mathrm{A}}\boldsymbol{\omega}^{\mathrm{B}} \tag{A.7}
$$

The translational velocity of the mass centers A and B and point C is given as

$$
\mathbf{V}_A = \frac{d\mathbf{P}_{NA}}{dt} = {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \mathbf{P}_{NA} \tag{A.8}
$$

$$
\mathbf{V}_C = \frac{d\mathbf{P}_{NC}}{dt} = {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \mathbf{P}_{NC} \tag{A.9}
$$

$$
\mathbf{V}_B = \frac{d\mathbf{P}_{NB}}{dt} = {}^{A}_{B}R^{T}\mathbf{V}_C + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{B}} \times \mathbf{P}_{CB} \tag{A.10}
$$

The quasi-velocities required to compute partial derivatives in Kane's equations of motion are $\mathbf{u}_A$ and $\mathbf{u}_B$ which are already defined in Section 2.3.

Acceleration: The angular acceleration of body A and body B is

$$
\begin{aligned}
{}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}} &= 2(q_1\ddot{q}_2 - q_2\ddot{q}_1 - q_3\ddot{q}_4 + q_4\ddot{q}_3)\widehat{\mathbf{A}}_1 \\
&\quad + 2(q_1\ddot{q}_3 + q_2\ddot{q}_4 - q_3\ddot{q}_1 - q_4\ddot{q}_2)\widehat{\mathbf{A}}_2 \\
&\quad + 2(q_1\ddot{q}_4 - q_2\ddot{q}_3 + q_3\ddot{q}_2 - q_4\ddot{q}_1)\widehat{\mathbf{A}}_3
\end{aligned} \tag{A.11}
$$

$$
\begin{aligned}
{}^{\mathrm{A}}\dot{\boldsymbol{\omega}}^{\mathrm{B}} &= 2(q_5\ddot{q}_6 - q_6\ddot{q}_5 - q_7\ddot{q}_8 + q_8\ddot{q}_7)\widehat{\mathbf{B}}_1 \\
&\quad + 2(q_5\ddot{q}_7 + q_6\ddot{q}_8 - q_7\ddot{q}_5 - q_8\ddot{q}_6)\widehat{\mathbf{B}}_2 \\
&\quad + 2(q_5\ddot{q}_8 - q_6\ddot{q}_7 + q_7\ddot{q}_6 - q_8\ddot{q}_5)\widehat{\mathbf{B}}_3
\end{aligned} \tag{A.12}
$$

$$
{}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{B}} = {}^{A}_{B}R^{T}\,{}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}} + {}^{\mathrm{A}}\dot{\boldsymbol{\omega}}^{\mathrm{B}} + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{B}} \times {}^{\mathrm{A}}\boldsymbol{\omega}^{\mathrm{B}} \tag{A.13}
$$

The translational acceleration of mass centers A and B and point C is given as

$$
\dot{\mathbf{V}}_A = \frac{d\mathbf{V}_A}{dt} = {}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}} \times \mathbf{P}_{NA} + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \left({}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \mathbf{P}_{NA}\right) \tag{A.14}
$$

$$
\dot{\mathbf{V}}_C = \frac{d\mathbf{V}_C}{dt} = {}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{A}} \times \mathbf{P}_{NC} + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \left({}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{A}} \times \mathbf{P}_{NC}\right) \tag{A.15}
$$

$$
\dot{\mathbf{V}}_B = \frac{d\mathbf{V}_B}{dt} = {}^{A}_{B}R^{T}\dot{\mathbf{V}}_C + {}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{B}} \times \mathbf{P}_{CB} + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{B}} \times \left({}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{B}} \times \mathbf{P}_{CB}\right) \tag{A.16}
$$

Mass Properties: Mass of body A and body B is assumed to be m_A and m_B respectively while the spherical joint is assumed to be massless. The inertia matrix of body A and body B can be given as

$$
I_{\mathrm{AA}} = \begin{bmatrix} \frac{1}{2}m_A r^2 & 0 & 0 \\ 0 & \frac{1}{12}m_A(3r^2 + L^2) & 0 \\ 0 & 0 & \frac{1}{12}m_A(3r^2 + L^2) \end{bmatrix} \tag{A.17}
$$

$$
I_{\mathrm{BB}} = \begin{bmatrix} \frac{1}{2}m_B r^2 & 0 & 0 \\ 0 & \frac{1}{12}m_B(3r^2 + L^2) & 0 \\ 0 & 0 & \frac{1}{12}m_B(3r^2 + L^2) \end{bmatrix} \tag{A.18}
$$

Forces and Moments: No external moments are acting on the bodies. Only the gravitational force acts on both the bodies. Note that both resultant force and moment vectors given below are expressed in inertial frame. Moments were taken about the mass center of each body.

$$\mathbf{F}_A = -m_A g \widehat{\mathbf{N}}_3 \quad , \quad \mathbf{F}_B = -m_B g \widehat{\mathbf{N}}_3 \tag{A.19}$$

$$\mathbf{M}_{AA} = \mathbf{0} \quad , \quad \mathbf{M}_{BB} = \mathbf{0} \tag{A.20}$$

Equations of Motion: The equations of motion can be computed using the Kane's equations given below. All the necessary terms are already defined in this appendix.

$$\begin{aligned} 0 &= F_i - F_i^* \\ F_i &= \sum_{K=1}^{bodies} \left[\mathbf{F}_K \cdot \frac{\partial \mathbf{V}_K}{\partial u_i} + \mathbf{M}_{KK} \cdot \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}}}{\partial u_i} \right] \\ F_i^* &= \sum_{K=1}^{bodies} \left[m_K \dot{\mathbf{V}}_K \cdot \frac{\partial \mathbf{V}_K}{\partial u_i} + \dot{\mathbf{H}}_{KK} \cdot \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}}}{\partial u_i} \right] \end{aligned} \tag{A.21}$$

where $i = \{1, 2, 3, 5, 6, 7\}$ and,

$$\dot{\mathbf{H}}_{KK} = I_{\mathrm{KK}} {}^{\mathrm{N}}\dot{\boldsymbol{\omega}}^{\mathrm{K}} + {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}} \times \left(I_{\mathrm{KK}} {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}} \right) \tag{A.22}$$

A.2 Online Constraint Embedding Method

The virtual work done by the system can be calculated from (A.21) as follows

$$0 = \delta W = \sum_{i=1}^{n} \sum_{K=1}^{bodies} \left[\mathbf{F}_K \cdot \frac{\partial \mathbf{V}_K}{\partial u_i} - m_K \dot{\mathbf{V}}_K \cdot \frac{\partial \mathbf{V}_K}{\partial u_i} + \mathbf{M}_{KK} \cdot \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}}}{\partial u_i} - \dot{\mathbf{H}}_{KK} \cdot \frac{\partial {}^{\mathrm{N}}\boldsymbol{\omega}^{\mathrm{K}}}{\partial u_i} \right] \delta q_i$$

The equation above can also be expressed as

$$0 = \sum_{i=1}^{n} (F_i - F_i^*) \delta q_i \tag{A.23}$$

Here, the F_i and F_i^* are generated for both the dependent and independent generalized coordinates. Note that any holonomic constraint can be expressed in a form that is linear in the virtual displacements. Let's say that we are solving for a system that has five generalized coordinates and two constraints. Thus, the relationship between dependent and independent virtual displacements can be given as

$$\begin{bmatrix} \delta q_{D_4} \\ \delta q_{D_5} \end{bmatrix} = \begin{bmatrix} C_{41} & C_{42} & C_{43} \\ C_{51} & C_{52} & C_{53} \end{bmatrix} \begin{bmatrix} \delta q_{I_1} \\ \delta q_{I_2} \\ \delta q_{I_3} \end{bmatrix} \tag{A.24}$$

where, the subscripts 'D' and 'I' denote a dependent or independent virtual displacement respectively. Substituting the equation above in Eqn. (A.23) yields

$$\begin{aligned} 0 &= (F_1 - F_1^*)\delta q_1 + (F_2 - F_2^*)\delta q_2 + (F_3 - F_3^*)\delta q_3 \\ &\quad + (F_4 - F_4^*)\delta q_4 + (F_5 - F_5^*)\delta q_5 \\ &= (F_1 - F_1^*)\delta q_1 + (F_2 - F_2^*)\delta q_2 + (F_3 - F_3^*)\delta q_3 \\ &\quad + (F_4 - F_4^*)(C_{41}\delta q_1 + C_{42}\delta q_2 + C_{43}\delta q_3) \\ &\quad + (F_5 - F_5^*)(C_{51}\delta q_1 + C_{52}\delta q_2 + C_{53}\delta q_3) \\ &= (F_1 - F_1^* + C_{41}(F_4 - F_4^*) + C_{51}(F_5 - F_5^*))\delta q_1 \\ &\quad + (F_2 - F_2^* + C_{42}(F_4 - F_4^*) + C_{52}(F_5 - F_5^*))\delta q_2 \\ &\quad + (F_3 - F_3^* + C_{43}(F_4 - F_4^*) + C_{53}(F_5 - F_5^*))\delta q_3 \end{aligned} \tag{A.25}$$

Because the virtual displacements in the equation above are independent, the only way for the virtual work to be equal to zero for any values they may take, is when their coefficients are zero for all time. Thus, it can be shown that

$$\bar{F}_i - \bar{F}_i^* = 0 \qquad \bar{F}_i = F_i + \sum_{j=p+1}^{p+q} C_{ji} F_j \qquad \bar{F}_i^* = F_i^* + \sum_{j=p+1}^{p+q} C_{ji} F_j^* \tag{A.26}$$

where, p and q are the number of independent and dependent generalized coordinates respectively. And, $i = \{1, 2, 3, ..., p\}$.

APPENDIX B

PSD Profiles and Curve Fits

Throughout this appendix, the blue line (—) corresponds to the data generated using the imBinarize centroid detection method while the orange line (—) corresponds to the data generated using the kernelFit centroid detection method. The solid black line represents the curve fit of Eqn. (4.12) to the experimental data. The stiffness values estimated through curve fitting are provided in the caption of each figure.

B.1 Data for 2μm diameter particle

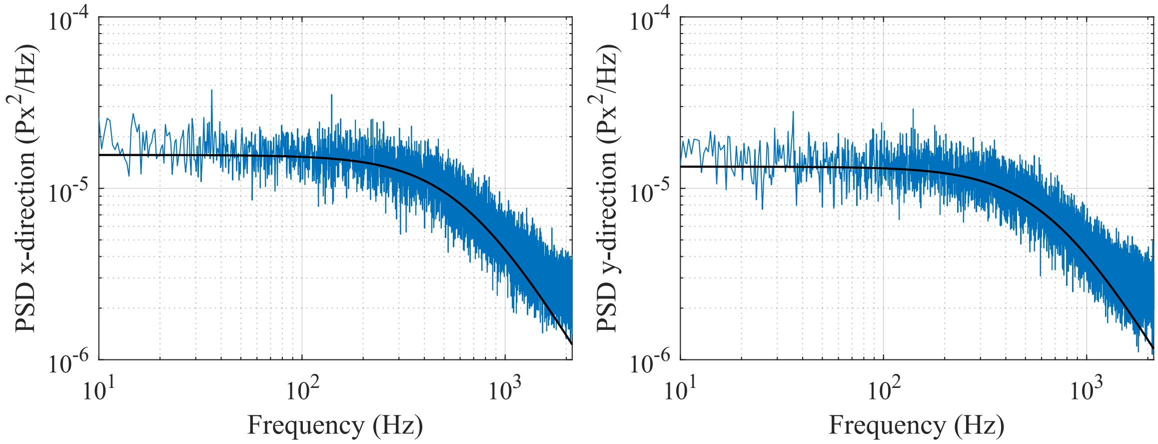


Figure B.1: $k_x = 73.9196$pN/μm, $k_y = 78.0766$pN/μm

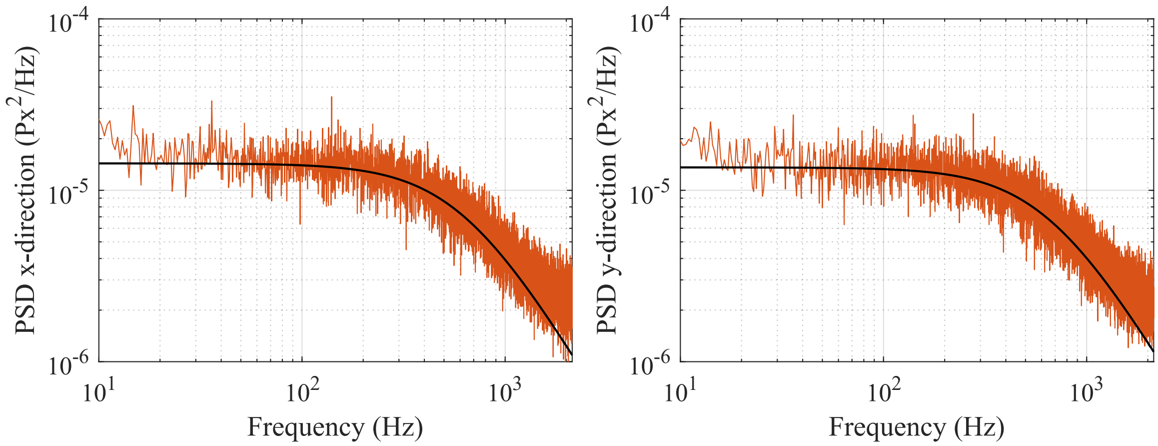


Figure B.2: $k_x = 72.7761$pN/μm, $k_y = 76.7711$pN/μm

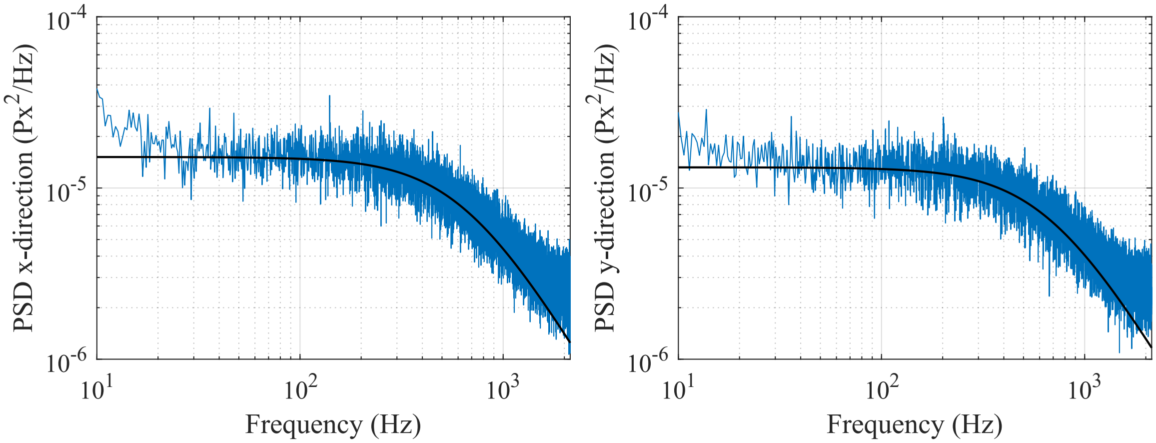


Figure B.3: $k_x = 75.8743\text{pN}/\mu\text{m}$, $k_y = 78.9234\text{pN}/\mu\text{m}$

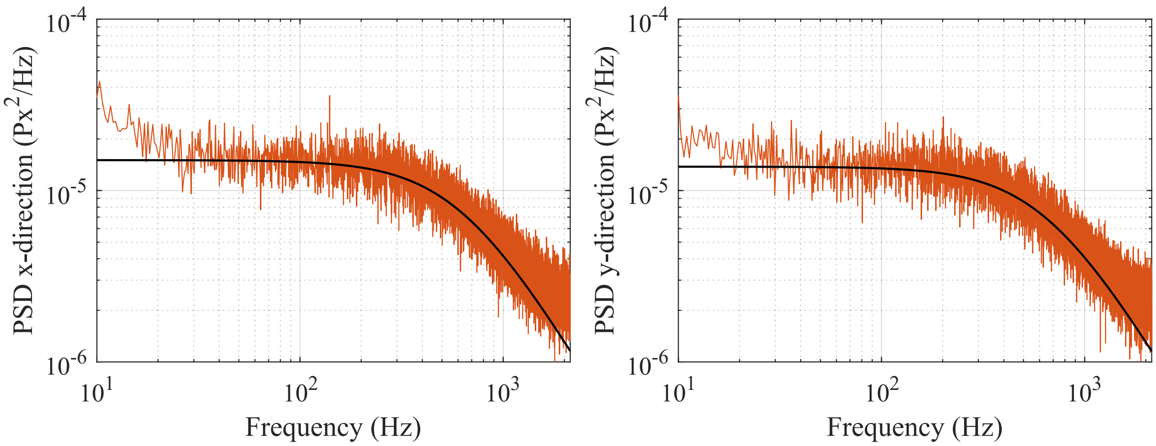


Figure B.4: $k_x = 73.2490\text{pN}/\mu\text{m}$, $k_y = 76.6792\text{pN}/\mu\text{m}$

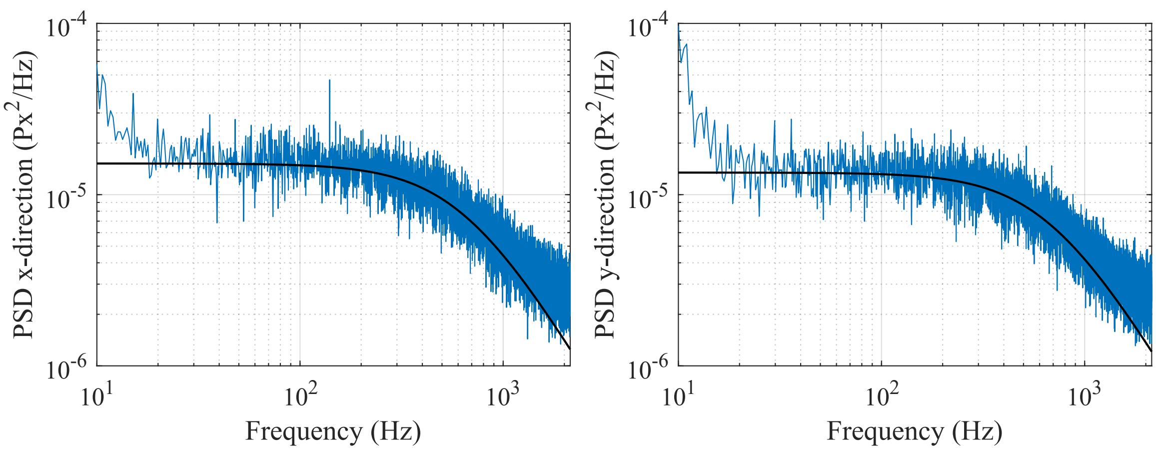


Figure B.5: $k_x = 75.8401\text{pN}/\mu\text{m}$, $k_y = 79.7704\text{pN}/\mu\text{m}$

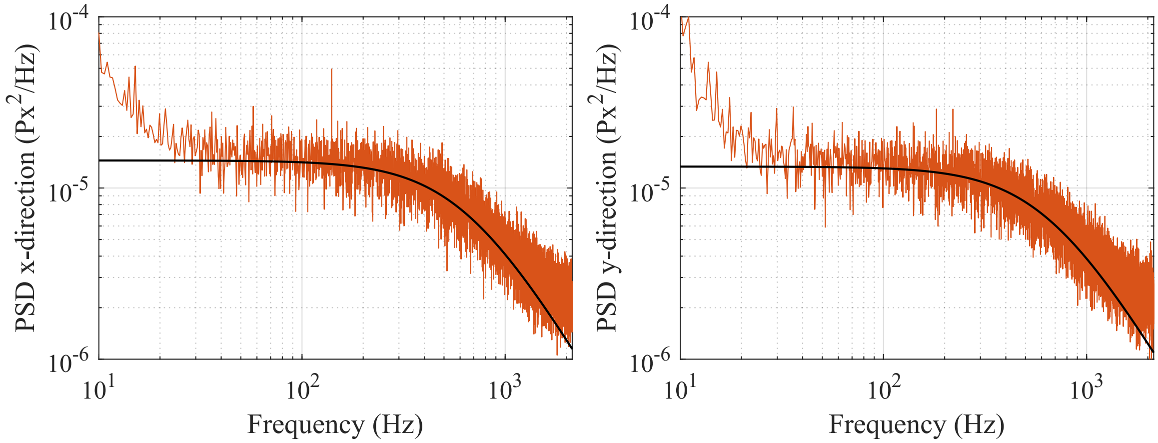


Figure B.6: $k_x = 74.4082 \text{pN}/\mu\text{m}$, $k_y = 75.7354 \text{pN}/\mu\text{m}$

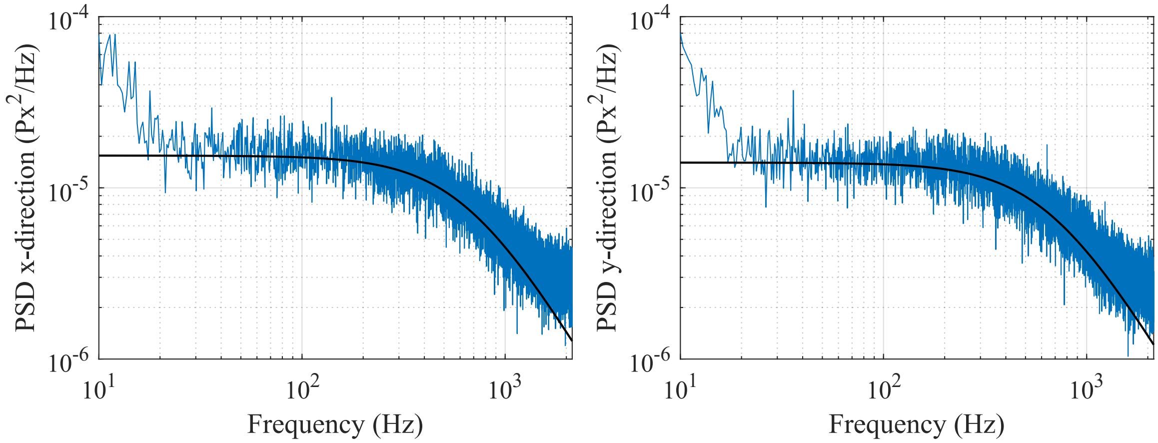


Figure B.7: $k_x = 76.0498 \text{pN}/\mu\text{m}$, $k_y = 77.9873 \text{pN}/\mu\text{m}$

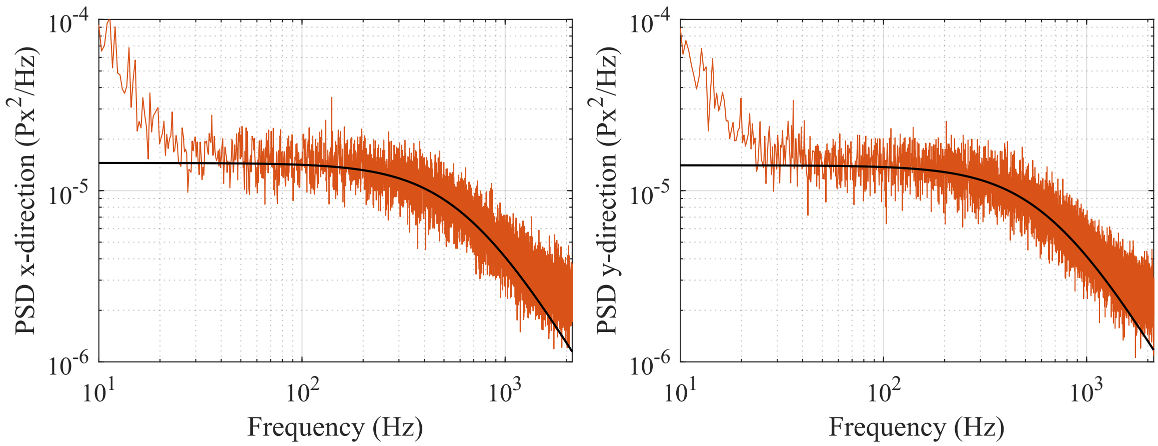


Figure B.8: $k_x = 74.2808 \text{pN}/\mu\text{m}$, $k_y = 76.4626 \text{pN}/\mu\text{m}$

B.2 Data for 1μm diameter particle

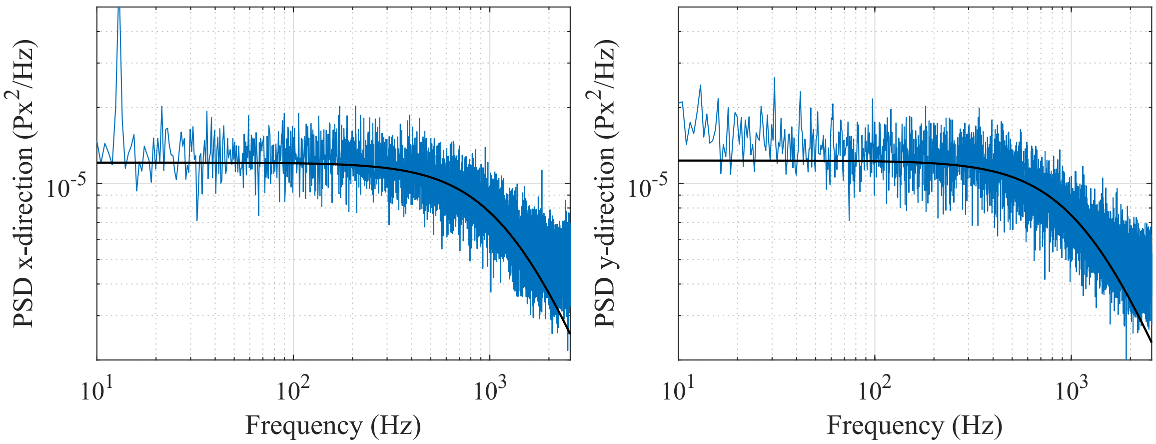


Figure B.9: $k_x = 78.2915\text{pN}/\mu\text{m}$, $k_y = 73.7166\text{pN}/\mu\text{m}$

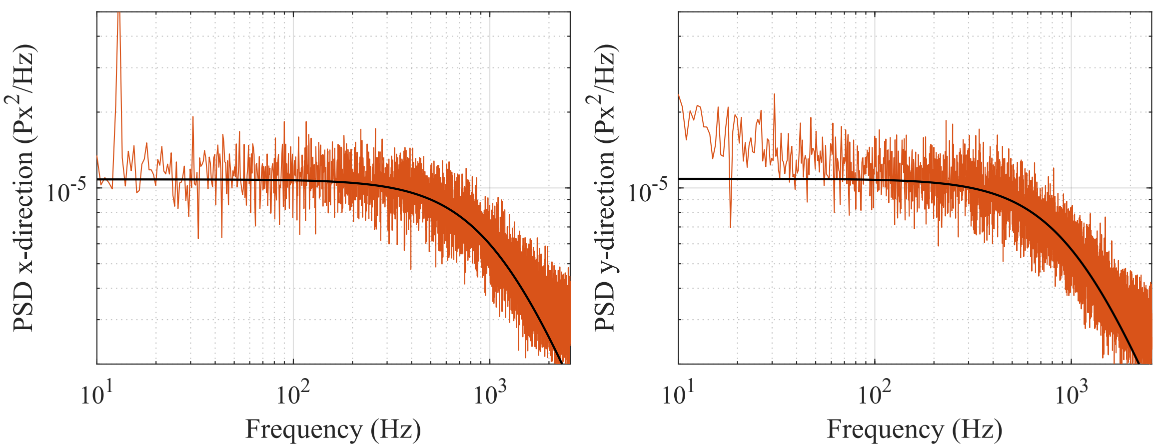


Figure B.10: $k_x = 65.9141\text{pN}/\mu\text{m}$, $k_y = 62.2912\text{pN}/\mu\text{m}$

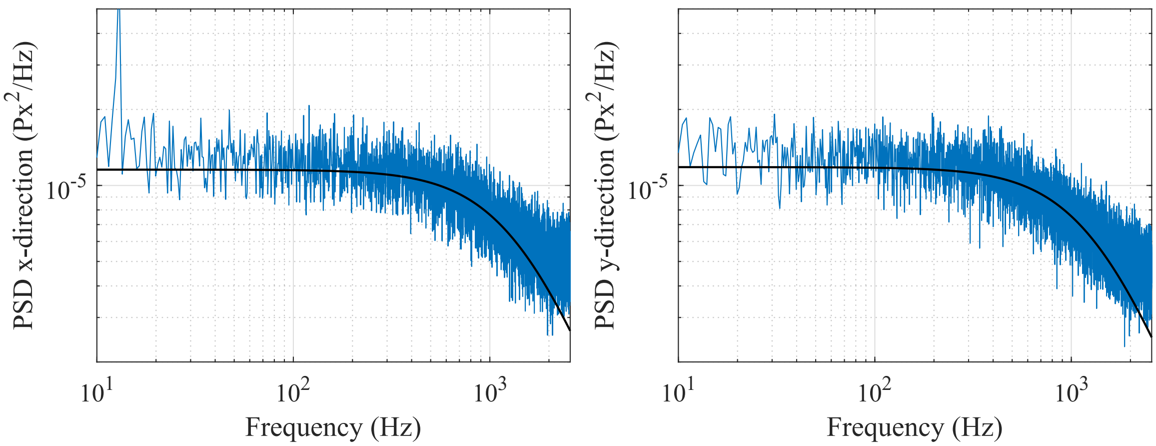


Figure B.11: $k_x = 83.1381 \text{pN}/\mu\text{m}$, $k_y = 78.8821 \text{pN}/\mu\text{m}$

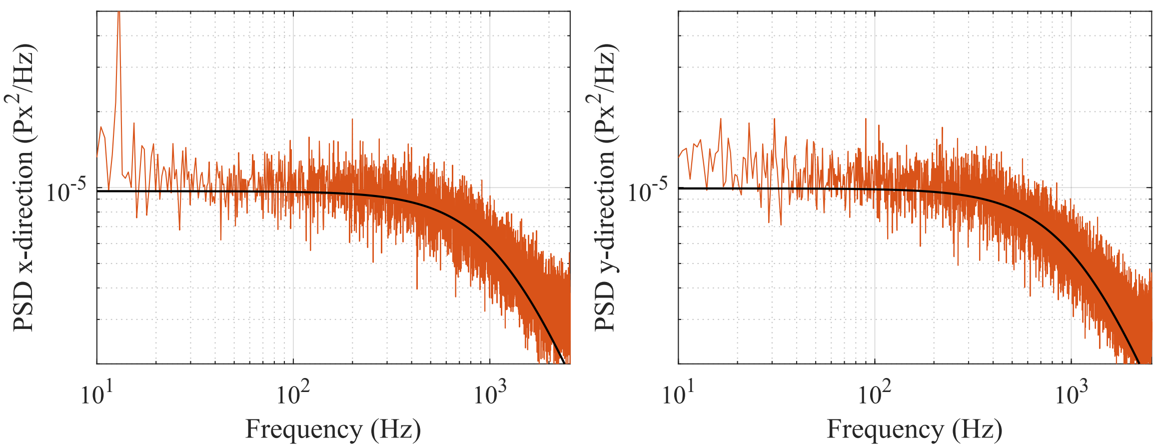


Figure B.12: $k_x = 72.6940 \text{pN}/\mu\text{m}$, $k_y = 66.1881 \text{pN}/\mu\text{m}$

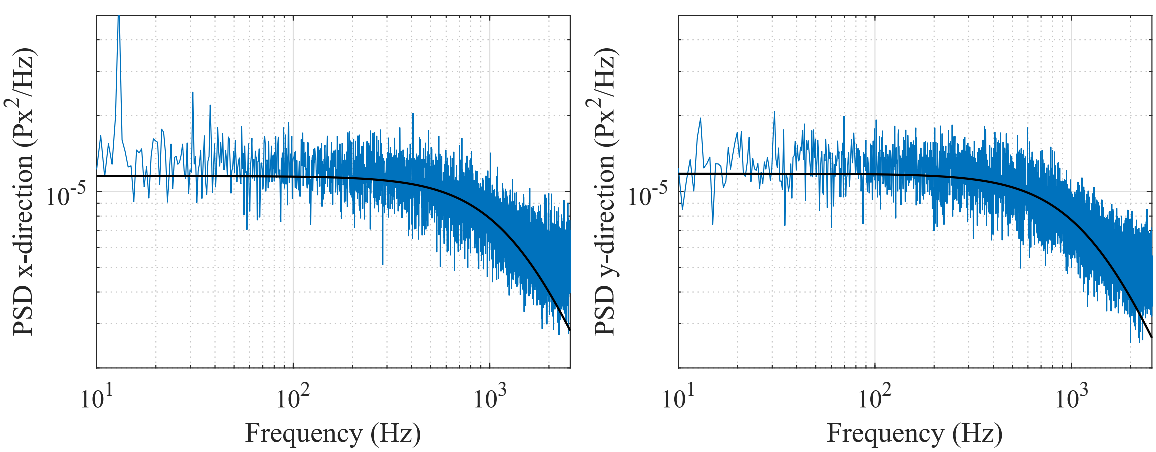


Figure B.13: $k_x = 86.4218 \text{pN}/\mu\text{m}$, $k_y = 81.6438 \text{pN}/\mu\text{m}$

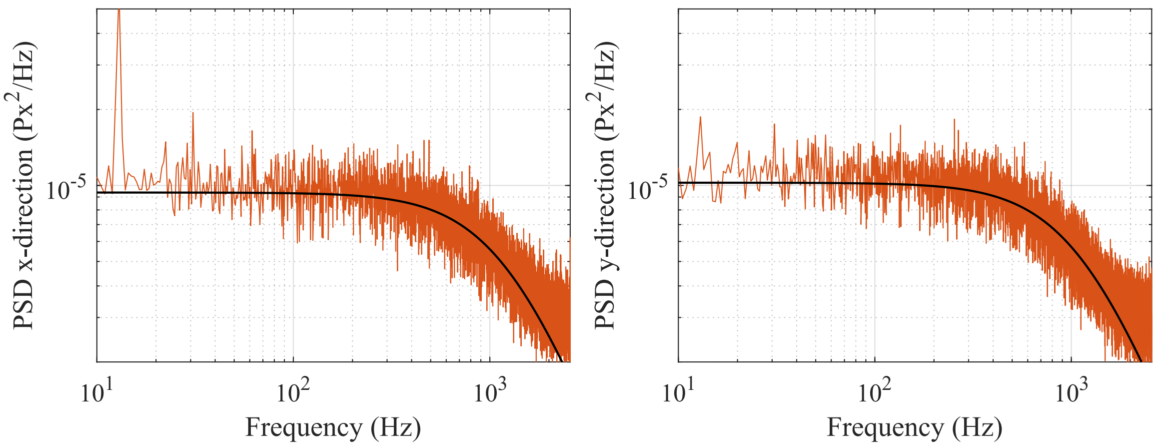


Figure B.14: $k_x = 72.0587\text{pN}/\mu\text{m}$, $k_y = 66.4809\text{pN}/\mu\text{m}$

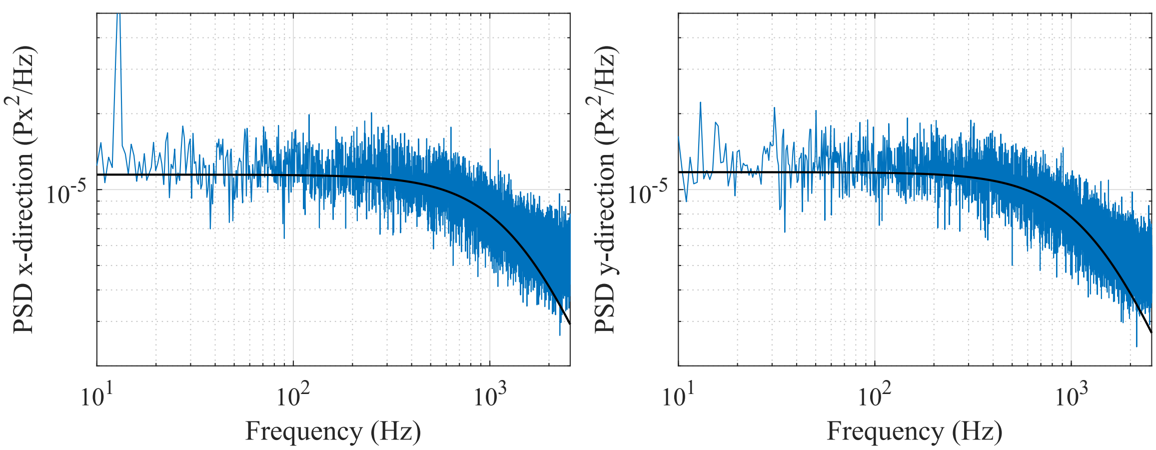


Figure B.15: $k_x = 88.9685\text{pN}/\mu\text{m}$, $k_y = 83.2548\text{pN}/\mu\text{m}$

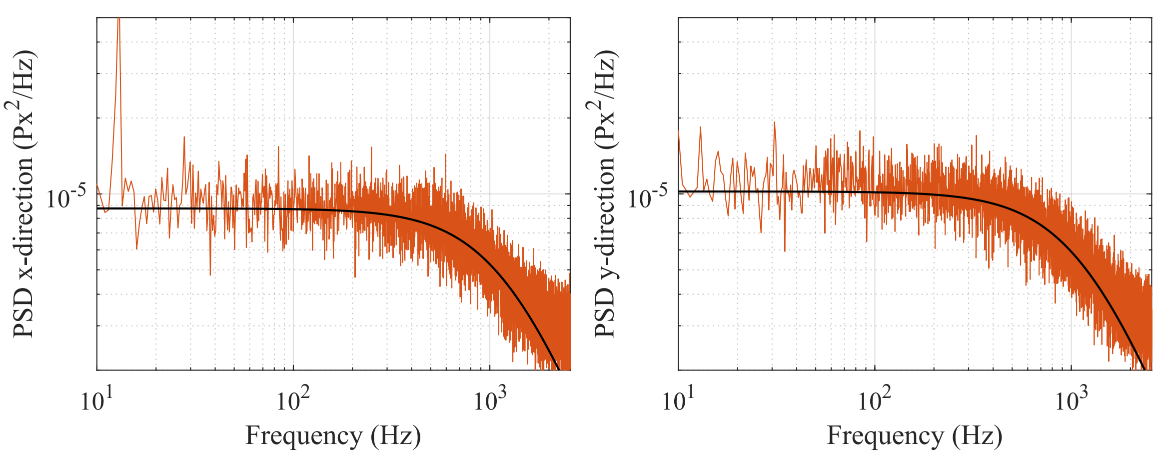


Figure B.16: $k_x = 72.7083\text{pN}/\mu\text{m}$, $k_y = 69.2532\text{pN}/\mu\text{m}$

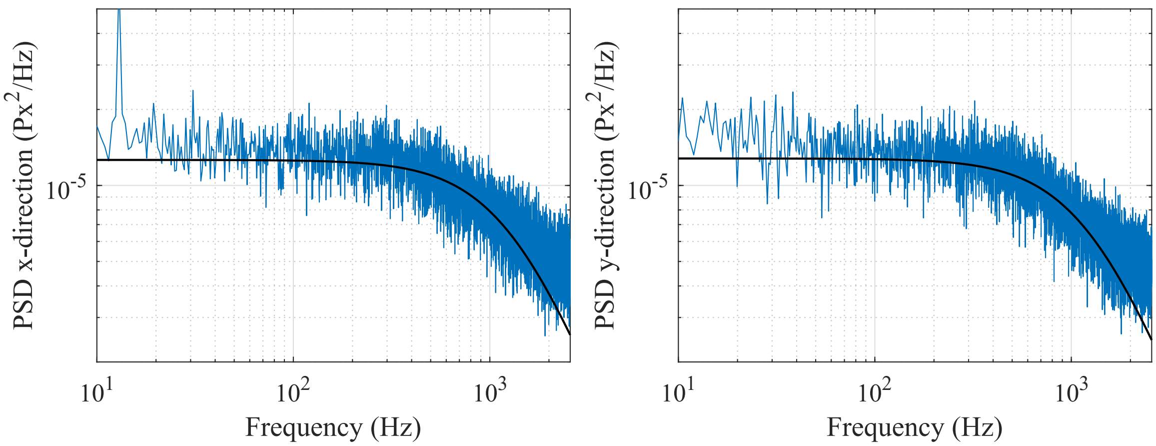


Figure B.17: $k_x = 76.7048 \text{pN}/\mu\text{m}$, $k_y = 73.9067 \text{pN}/\mu\text{m}$

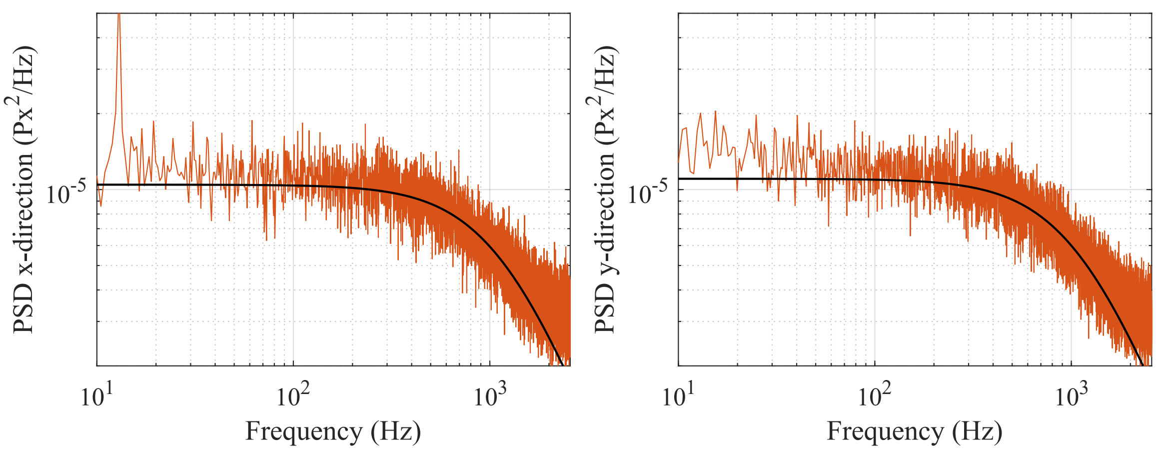


Figure B.18: $k_x = 68.0154 \text{pN}/\mu\text{m}$, $k_y = 64.6954 \text{pN}/\mu\text{m}$

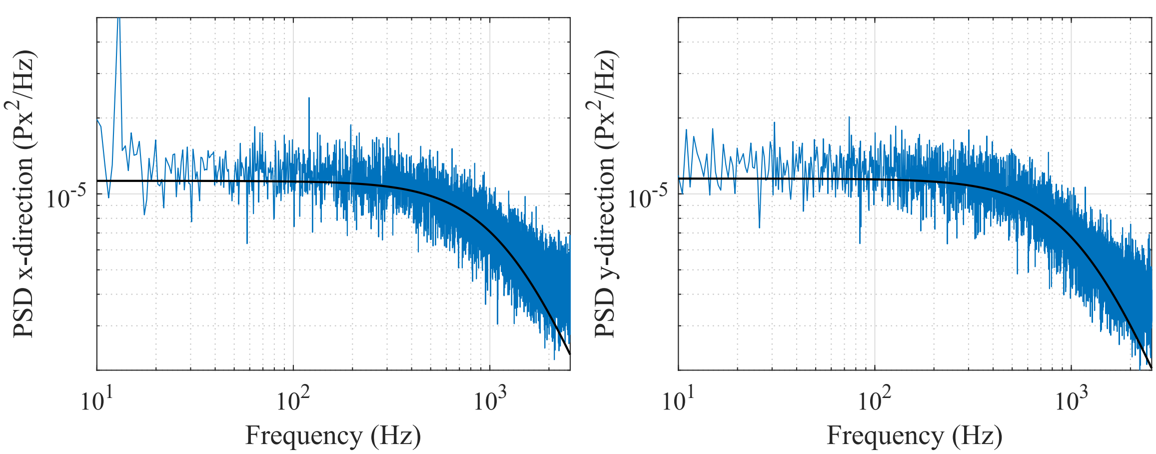


Figure B.19: $k_x = 77.1914 \text{pN}/\mu\text{m}$, $k_y = 70.5412 \text{pN}/\mu\text{m}$

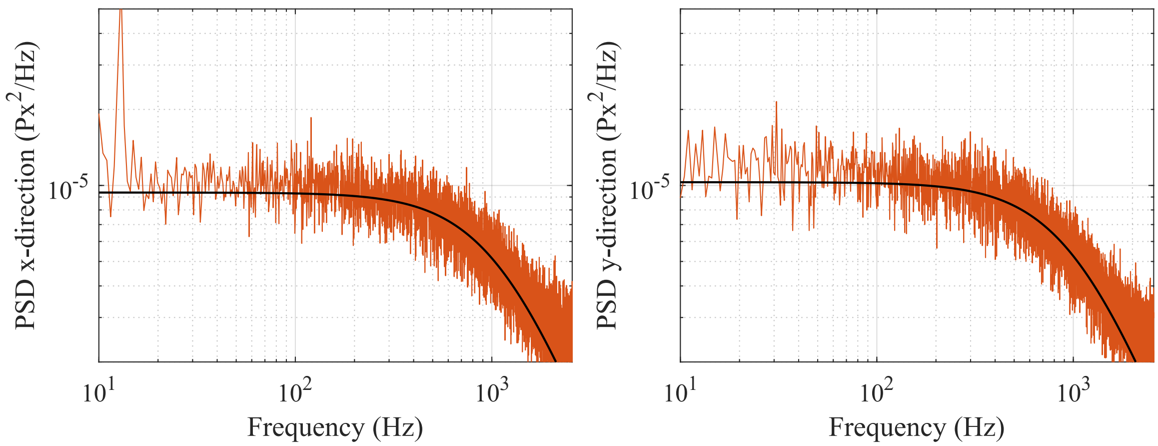


Figure B.20: $k_x = 65.5011$pN/μm, $k_y = 60.4925$pN/μm

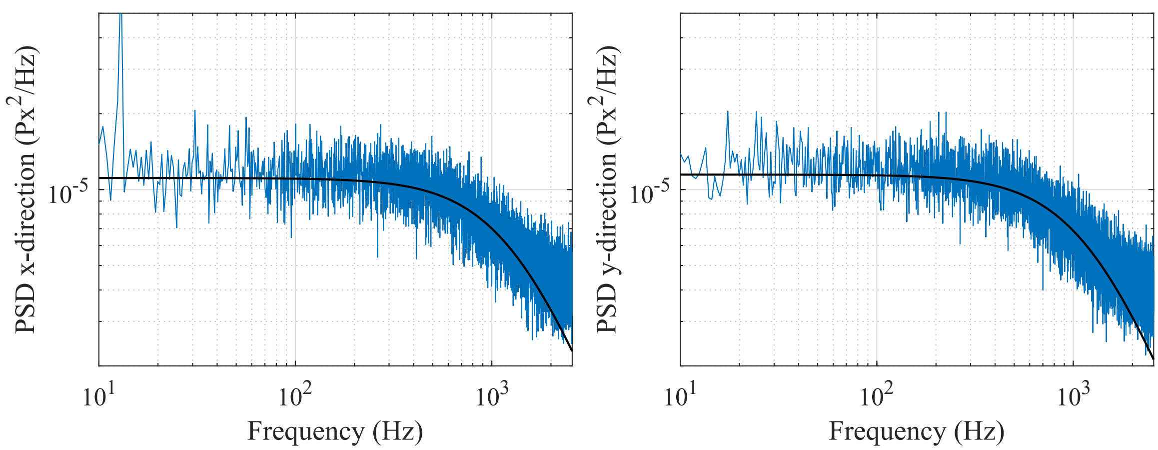


Figure B.21: $k_x = 77.4360$pN/μm, $k_y = 72.4398$pN/μm

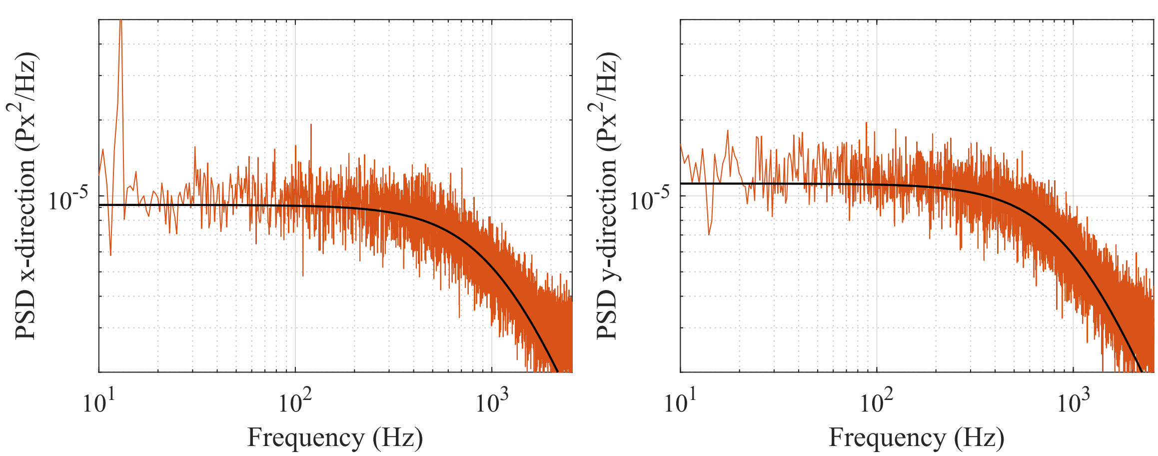


Figure B.22: $k_x = 67.7851$pN/μm, $k_y = 61.8914$pN/μm

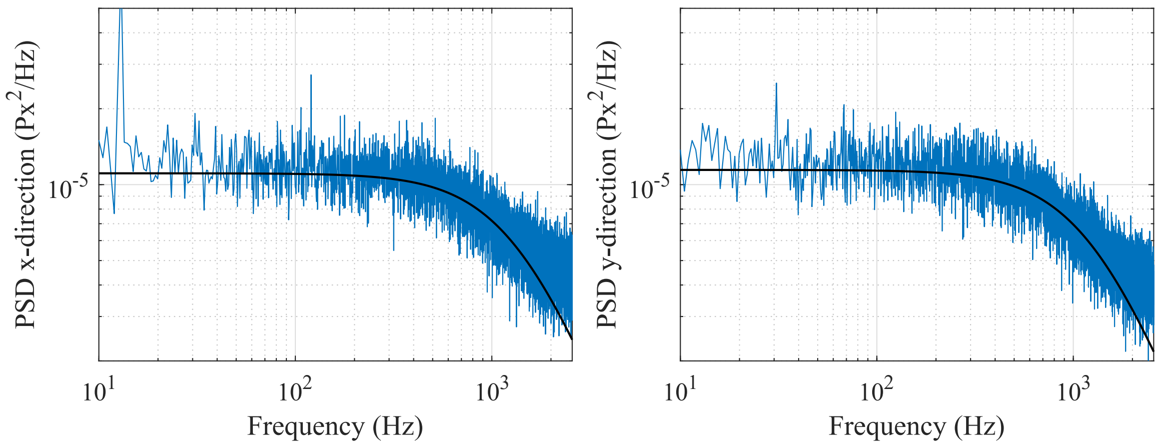


Figure B.23: $k_x = 80.5437$pN/μm, $k_y = 73.7746$pN/μm

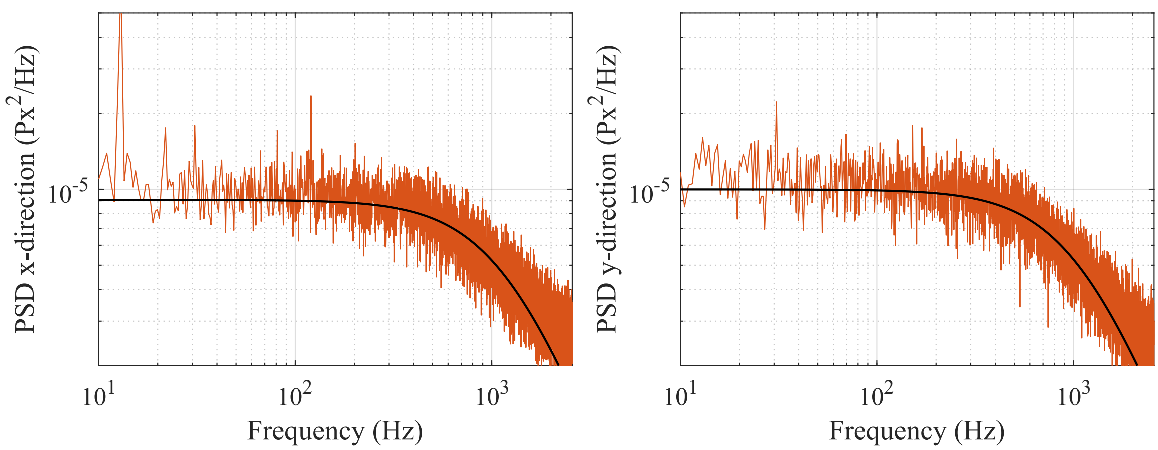


Figure B.24: $k_x = 69.0360$pN/μm, $k_y = 62.6550$pN/μm

B.3 Data for 500nmm diameter particle

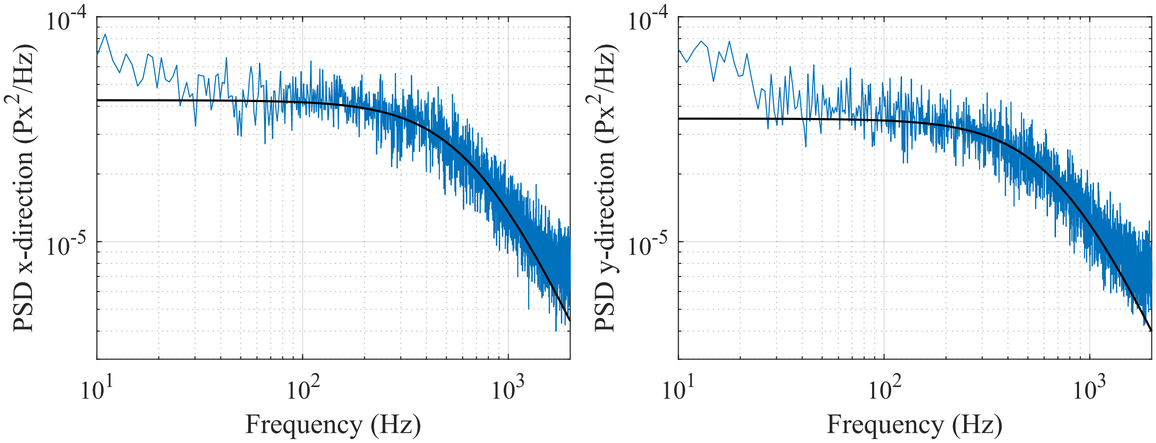


Figure B.25: $k_x = 20.2335\text{pN}/\mu\text{m}$, $k_y = 21.2079\text{pN}/\mu\text{m}$

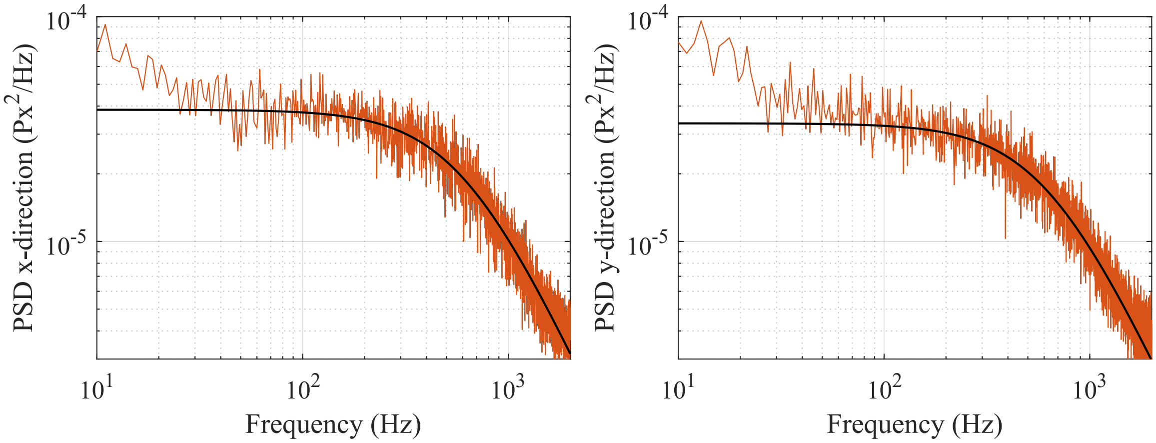


Figure B.26: $k_x = 17.7502\text{pN}/\mu\text{m}$, $k_y = 18.4455\text{pN}/\mu\text{m}$

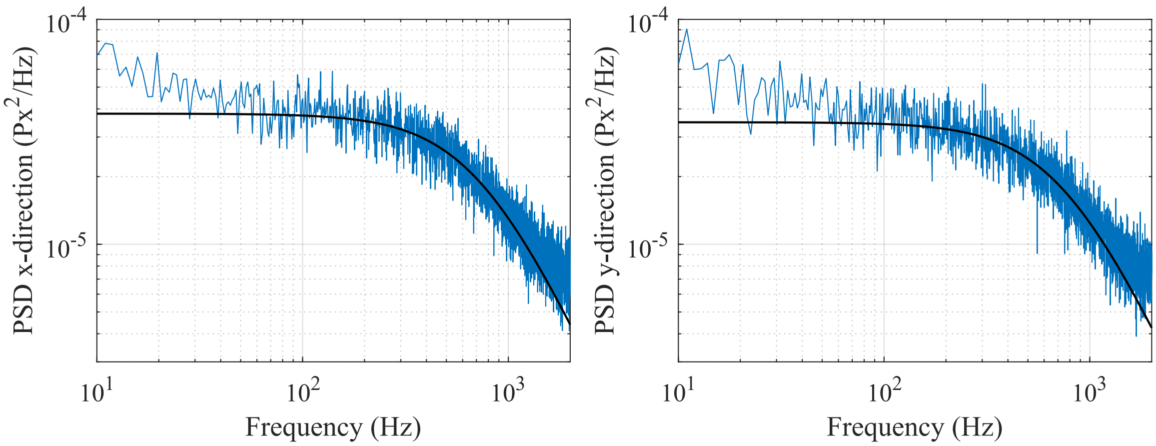


Figure B.27: $k_x = 21.4025\text{pN}/\mu\text{m}$, $k_y = 22.0832\text{pN}/\mu\text{m}$

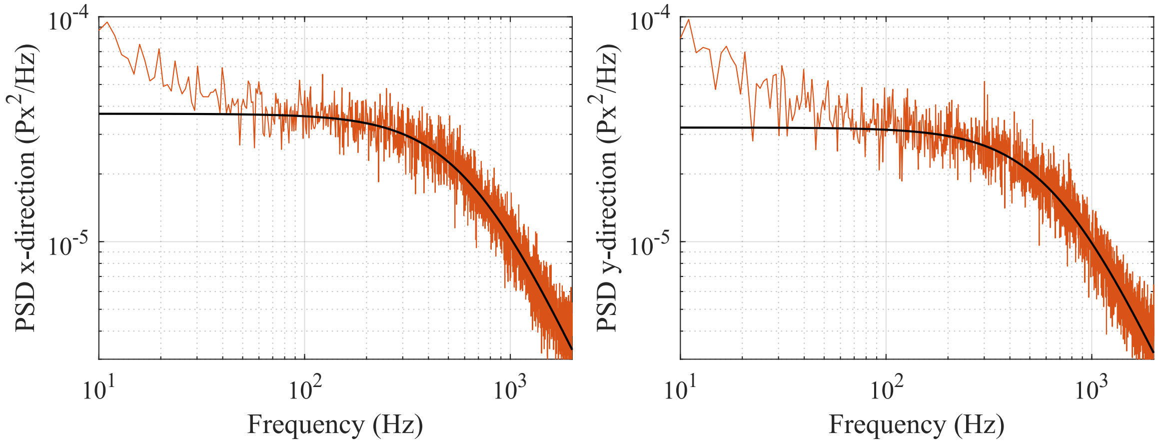


Figure B.28: $k_x = 18.5560\text{pN}/\mu\text{m}$, $k_y = 19.7081\text{pN}/\mu\text{m}$

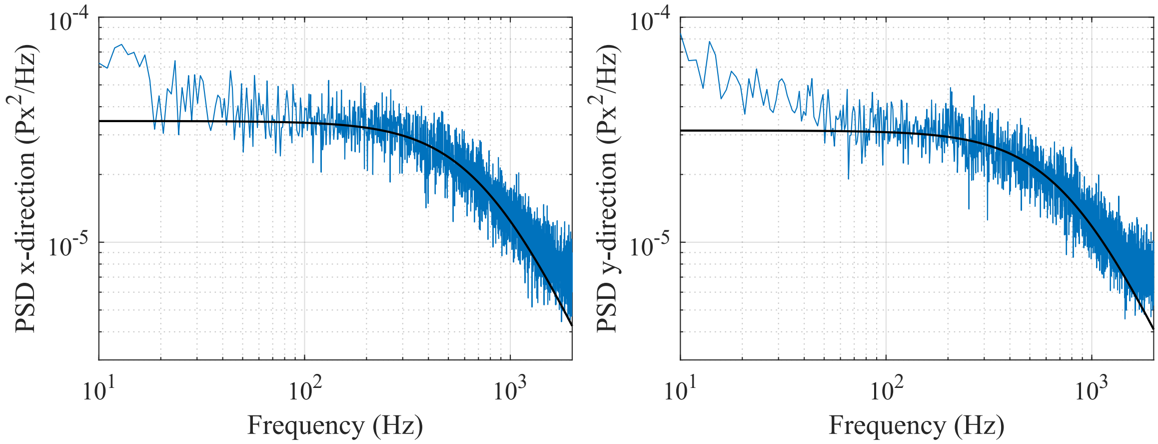


Figure B.29: $k_x = 22.2028\text{pN}/\mu\text{m}$, $k_y = 22.9997\text{pN}/\mu\text{m}$

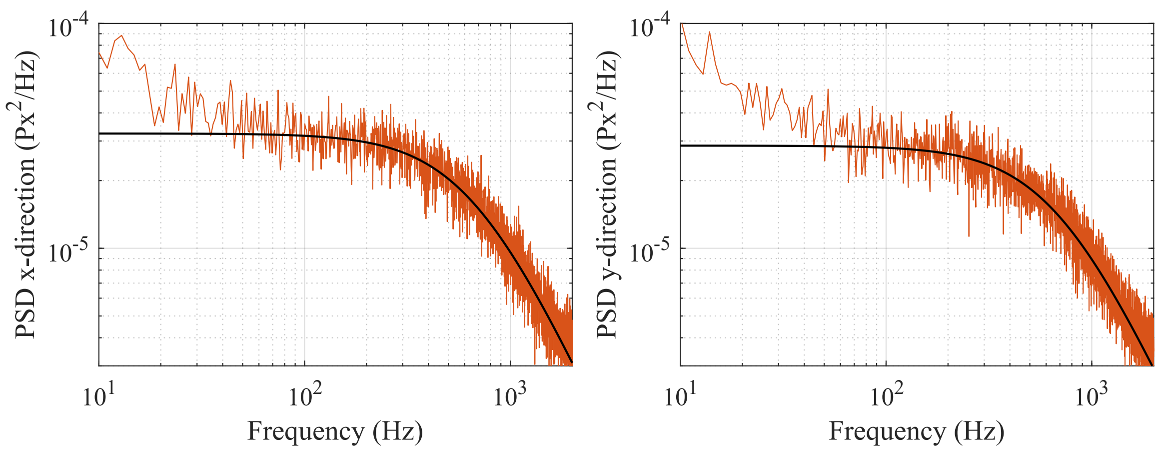


Figure B.30: $k_x = 19.2673\text{pN}/\mu\text{m}$, $k_y = 19.9635\text{pN}/\mu\text{m}$

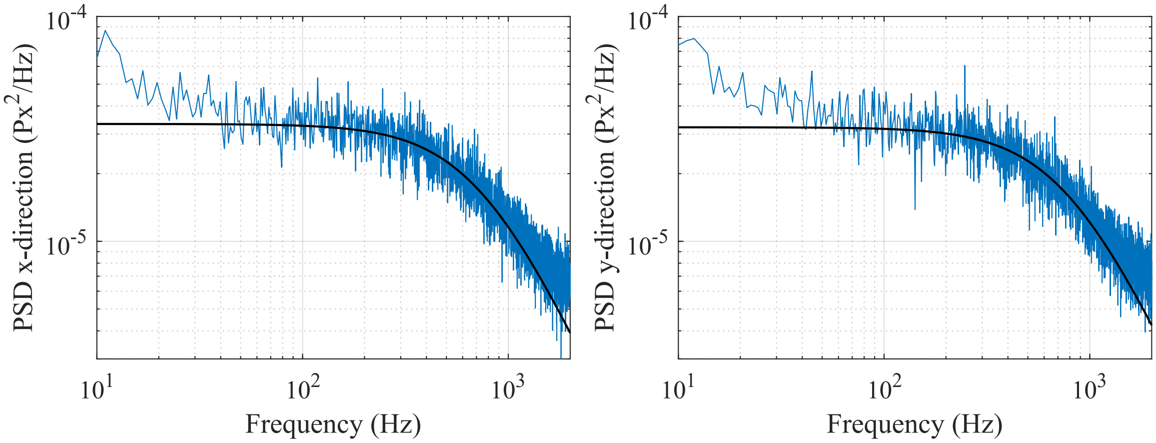


Figure B.31: $k_x = 21.6464\text{pN}/\mu\text{m}$, $k_y = 23.1000\text{pN}/\mu\text{m}$

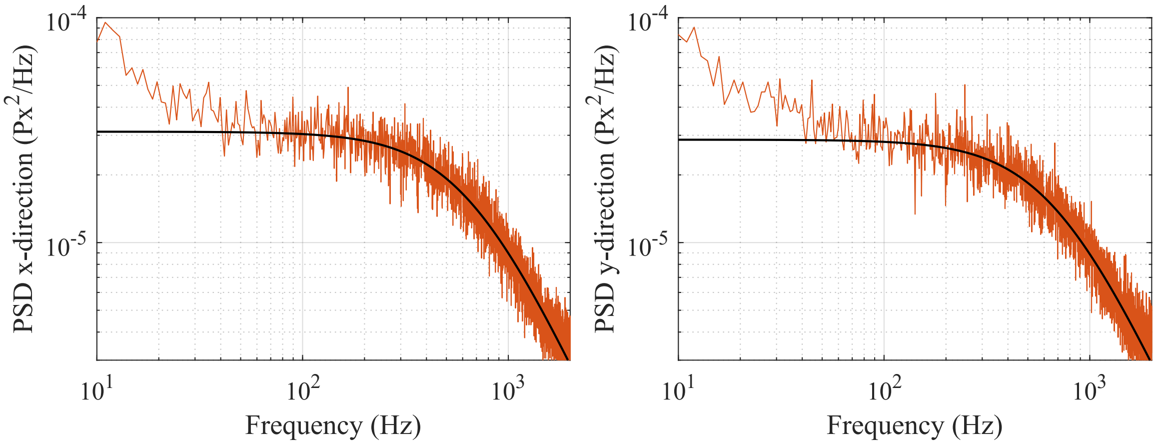
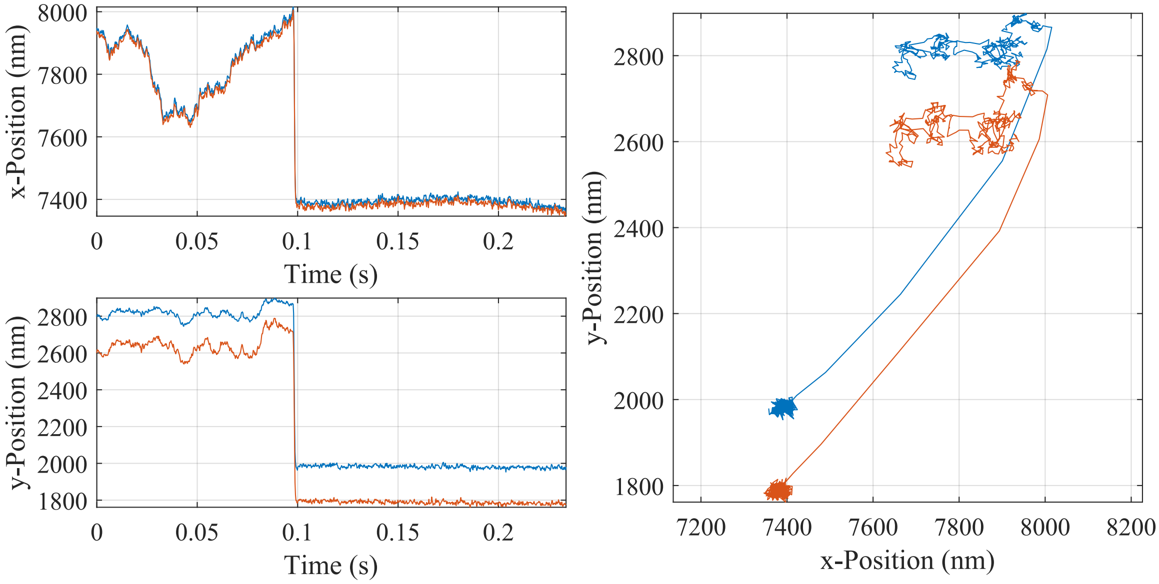
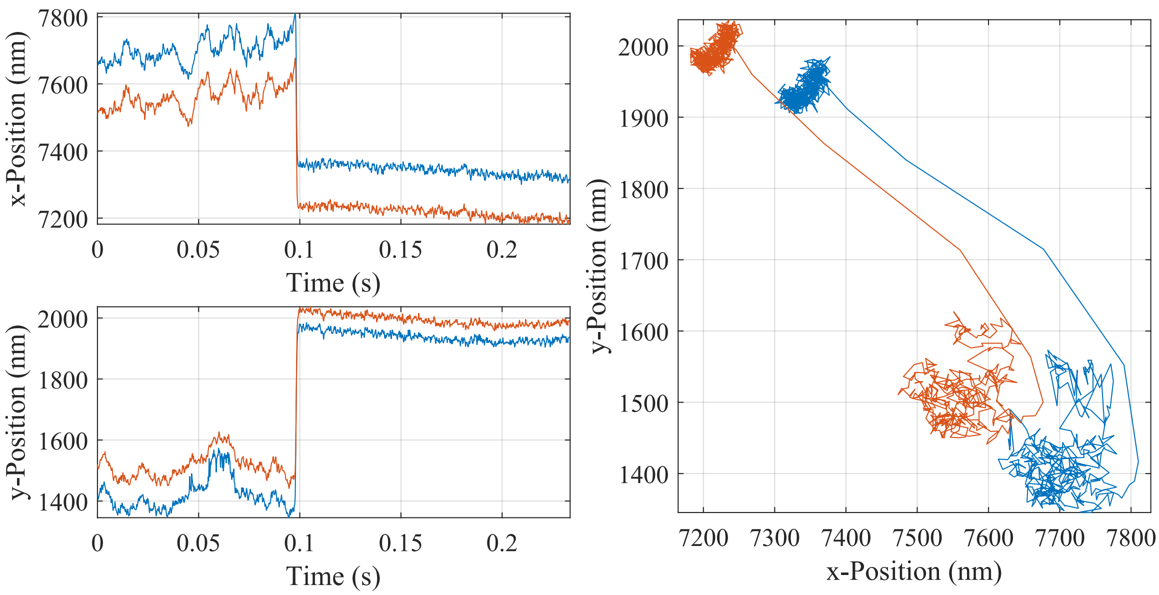


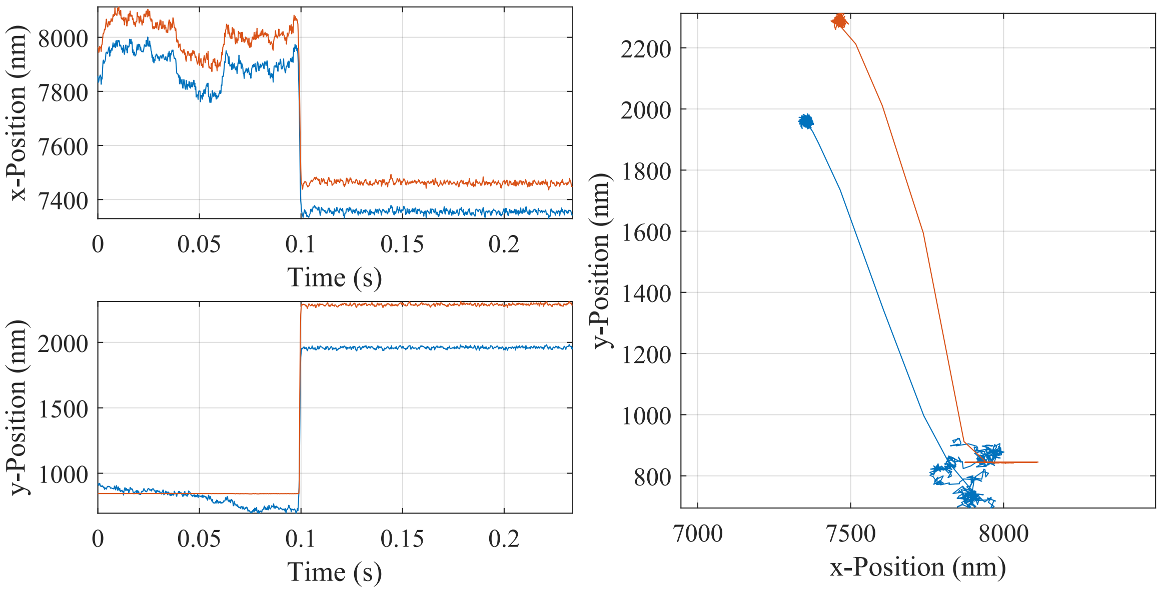
Figure B.32: $k_x = 18.8710\text{pN}/\mu\text{m}$, $k_y = 19.8701\text{pN}/\mu\text{m}$

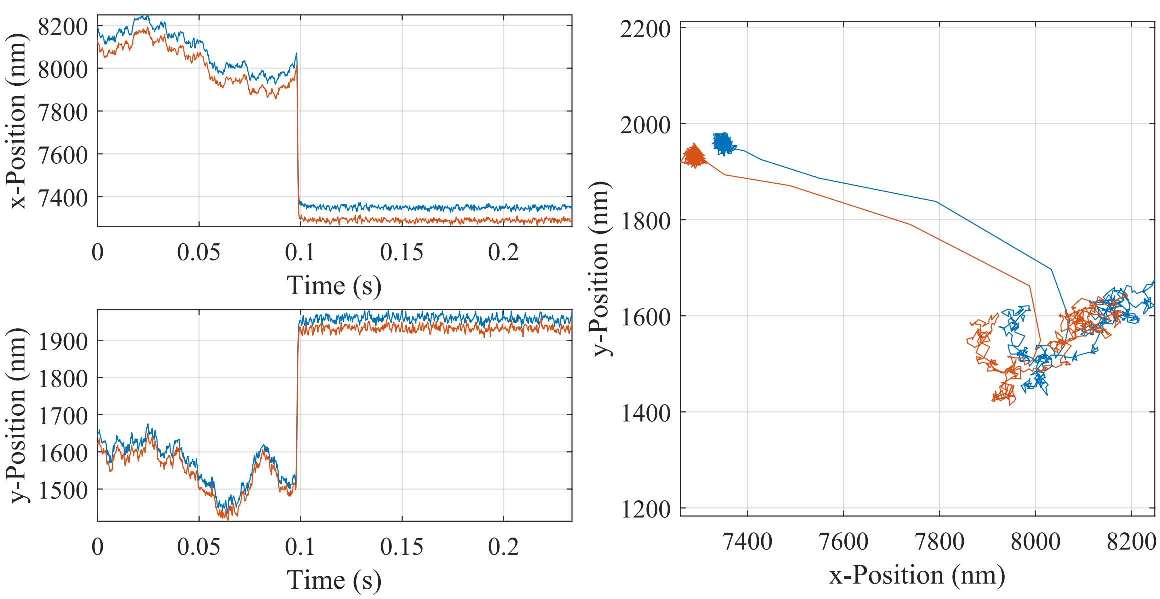
APPENDIX C

TRR Experiments Data for 2μm Diameter Particle


x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

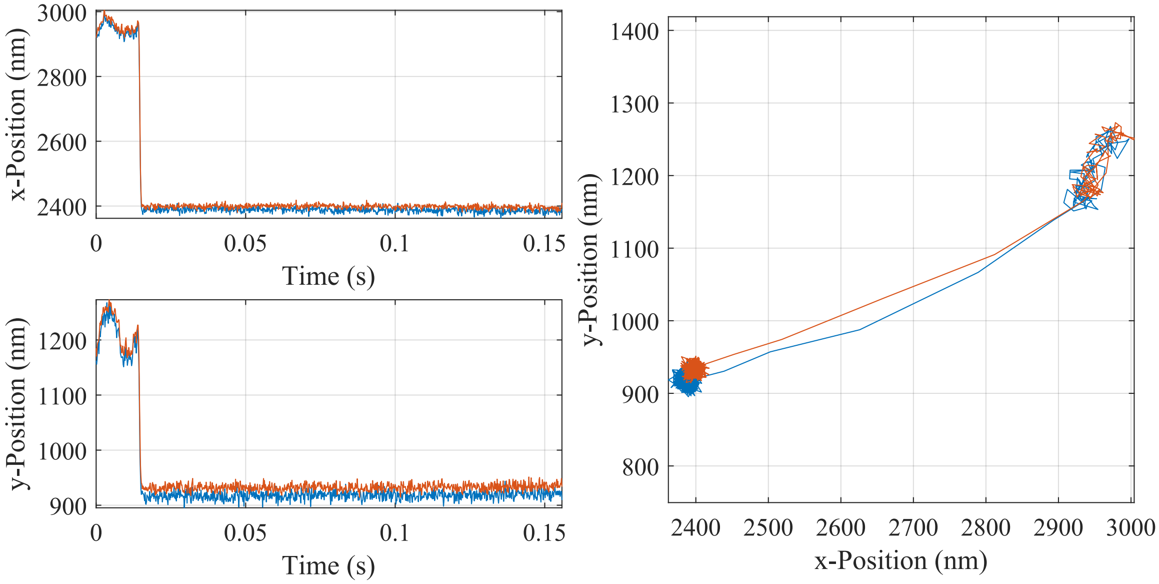

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

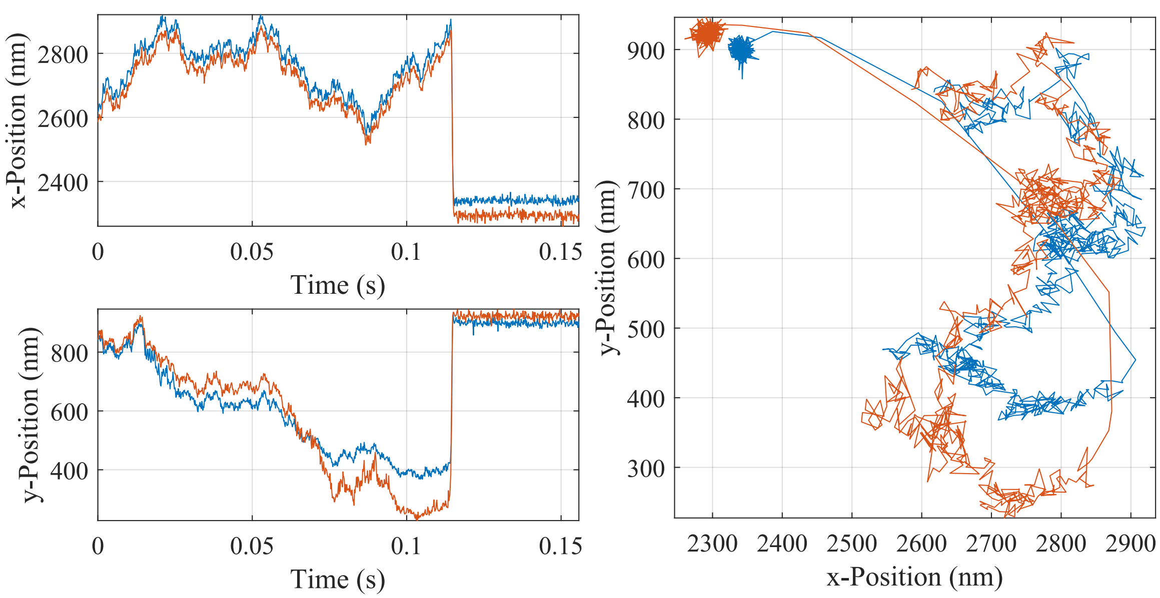

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

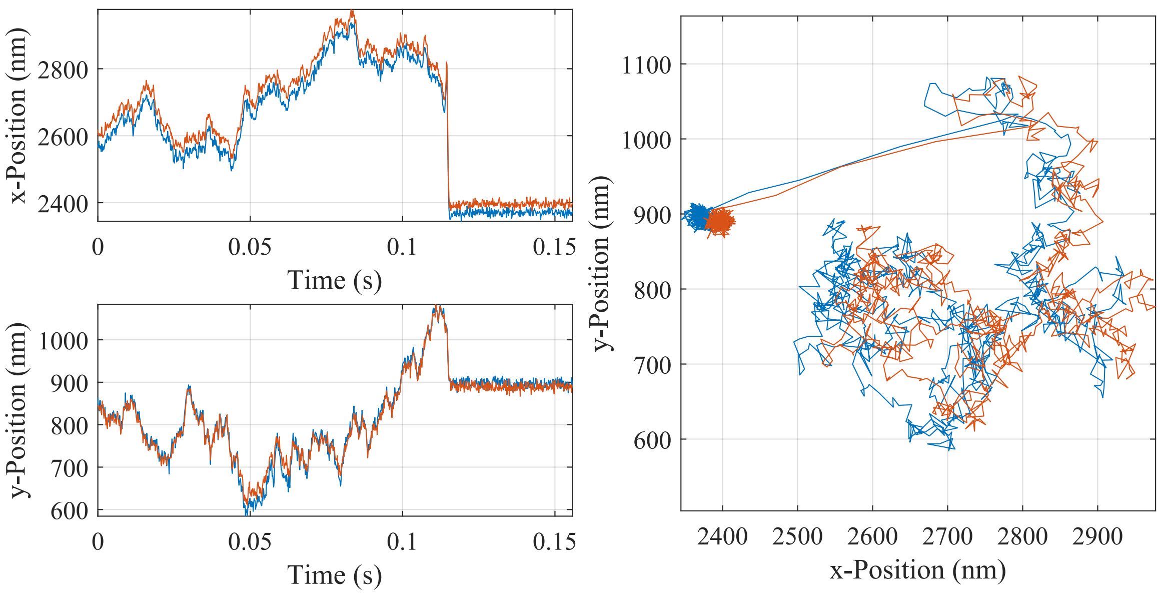

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

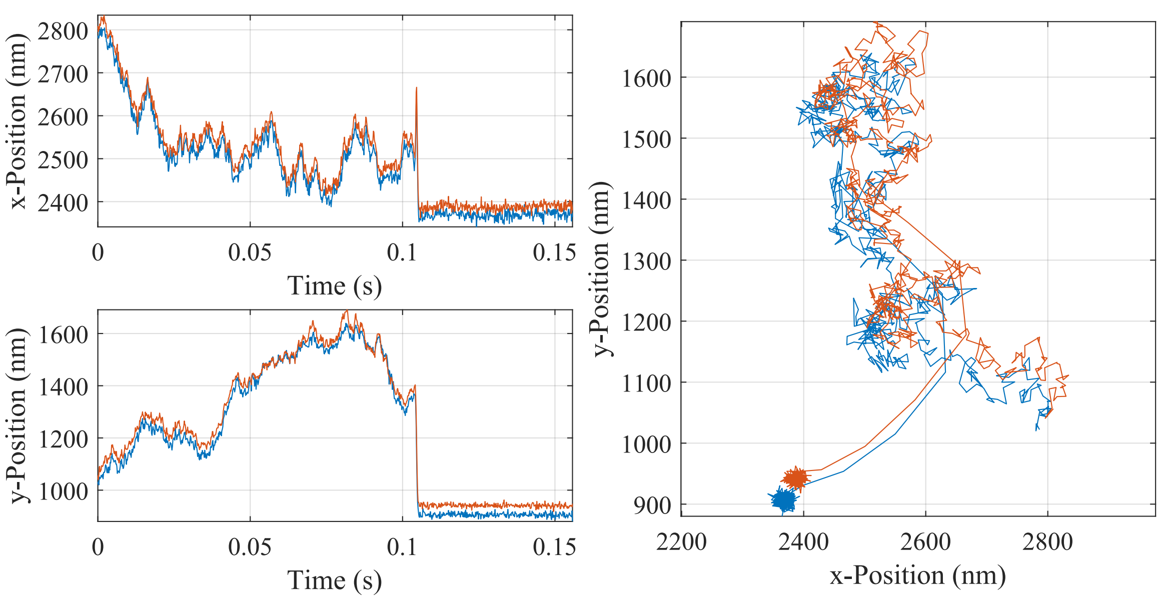
APPENDIX D

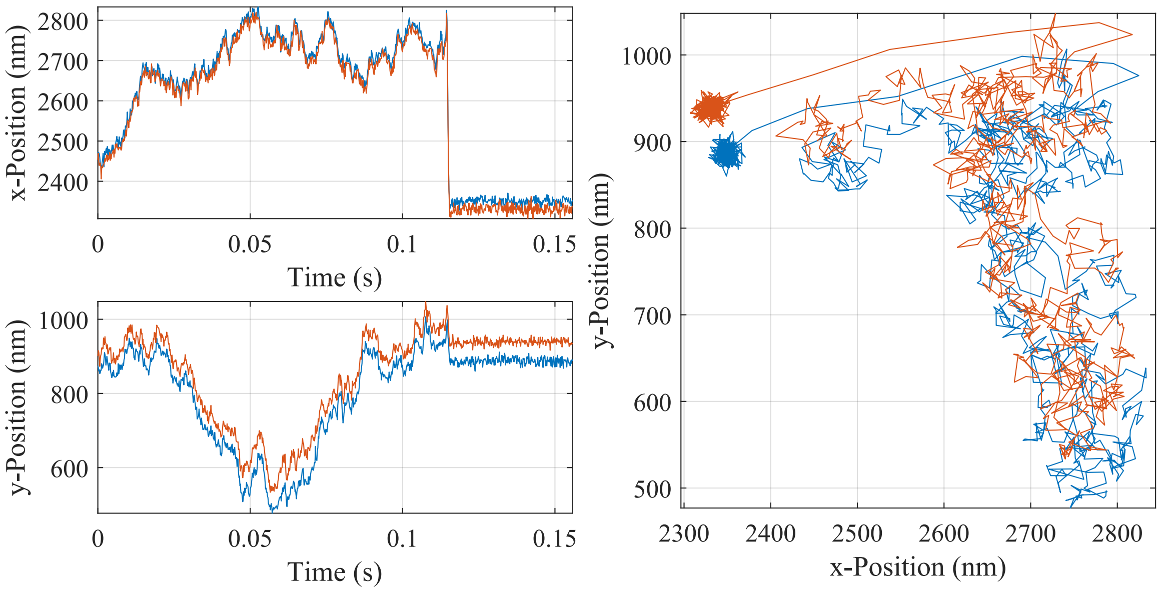
TRR Experiments Data for 1μm Diameter Particle

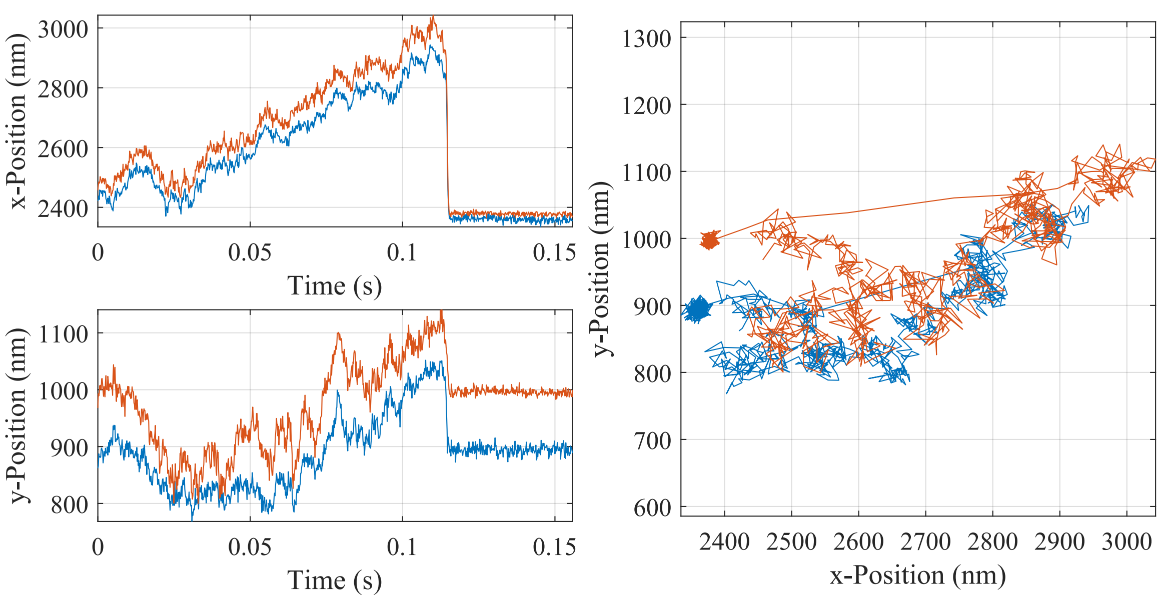

x-Position (nm)
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1300
1200
1100
1000
900
800
2400
2500
2600
2700
2800
2900
3000
x-Position (nm)

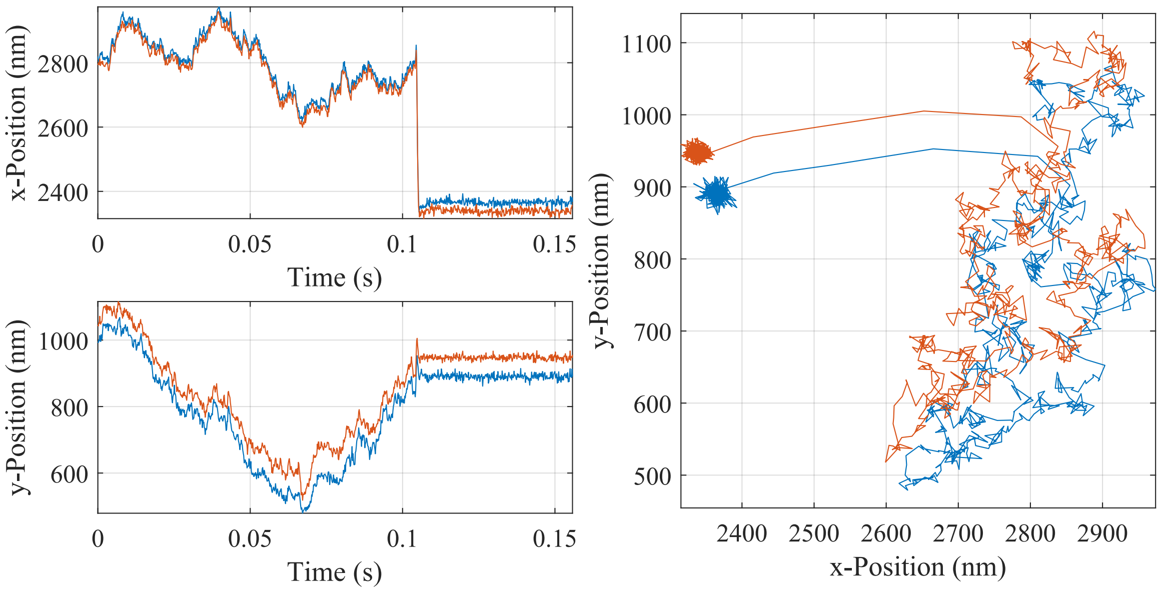

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
800
600
400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
900
800
700
600
500
400
300
2300
2400
2500
2600
2700
2800
2900
x-Position (nm)

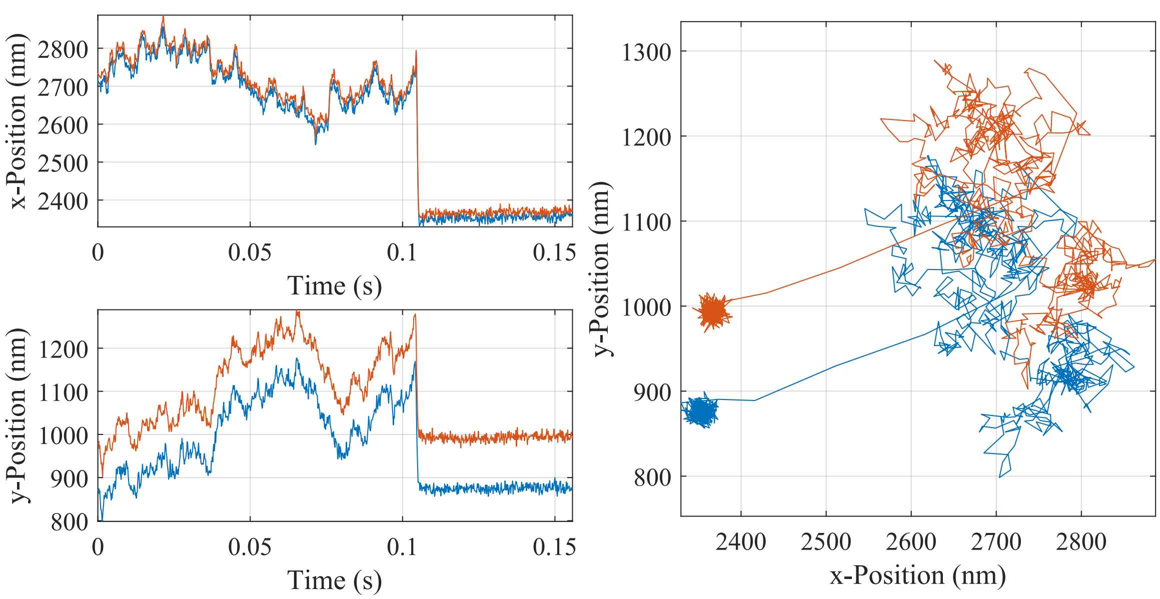

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

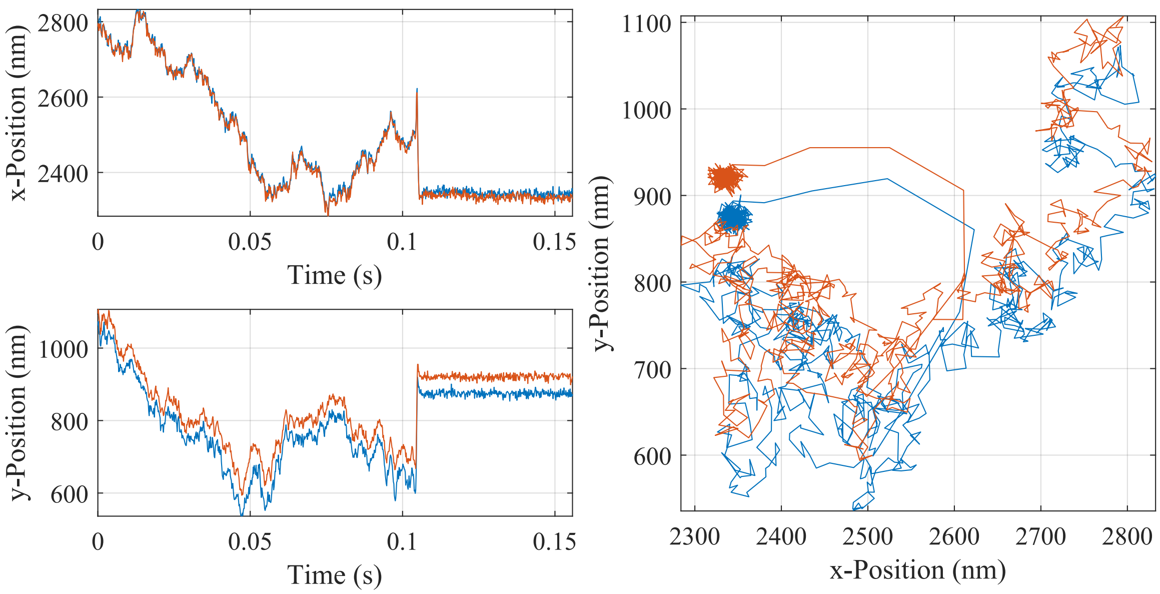

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

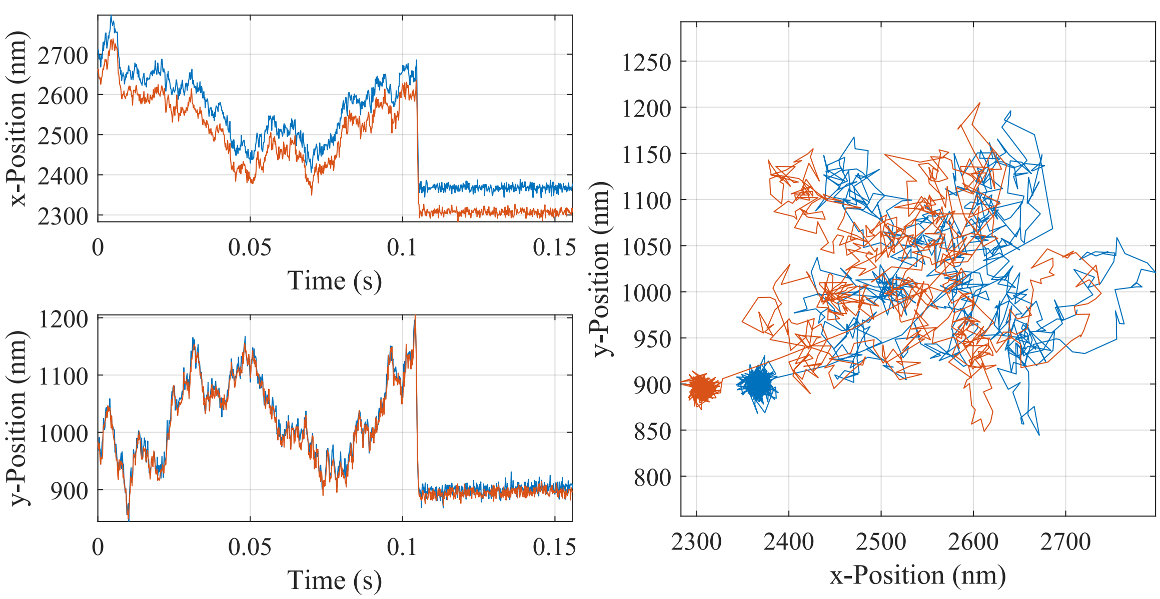

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

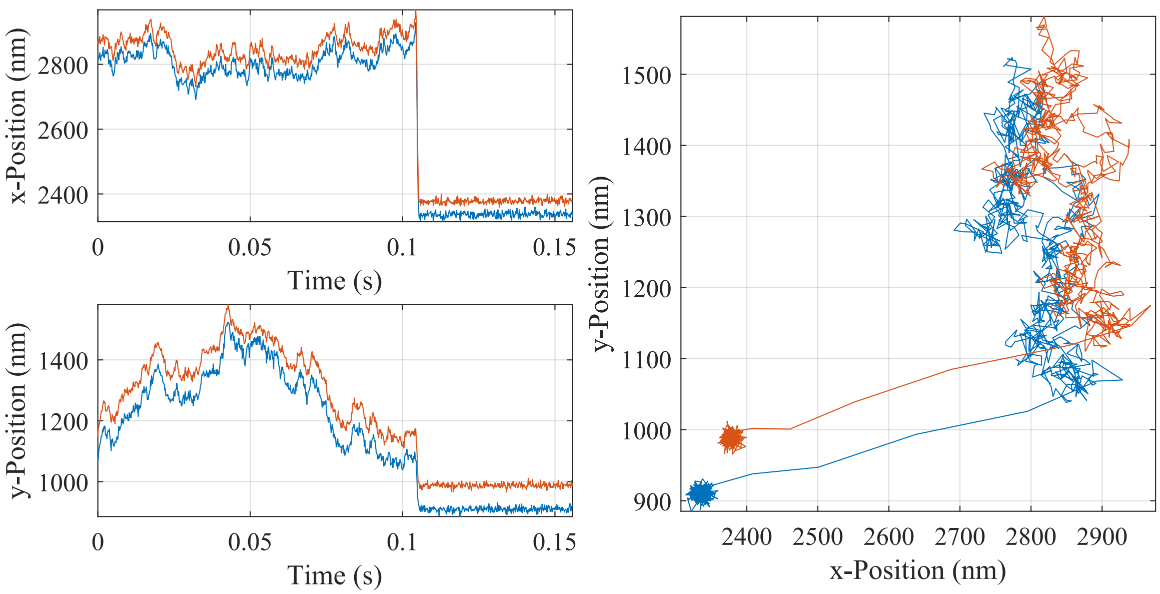

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

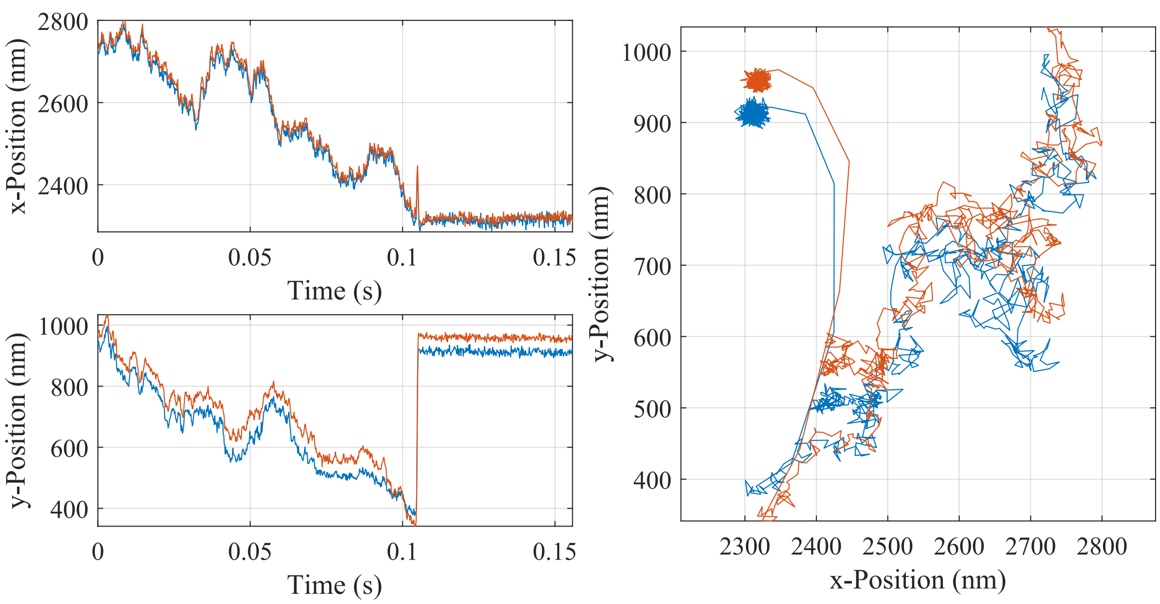

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
800
600
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
700
600
500
2400
2500
2600
2700
2800
2900
x-Position (nm)

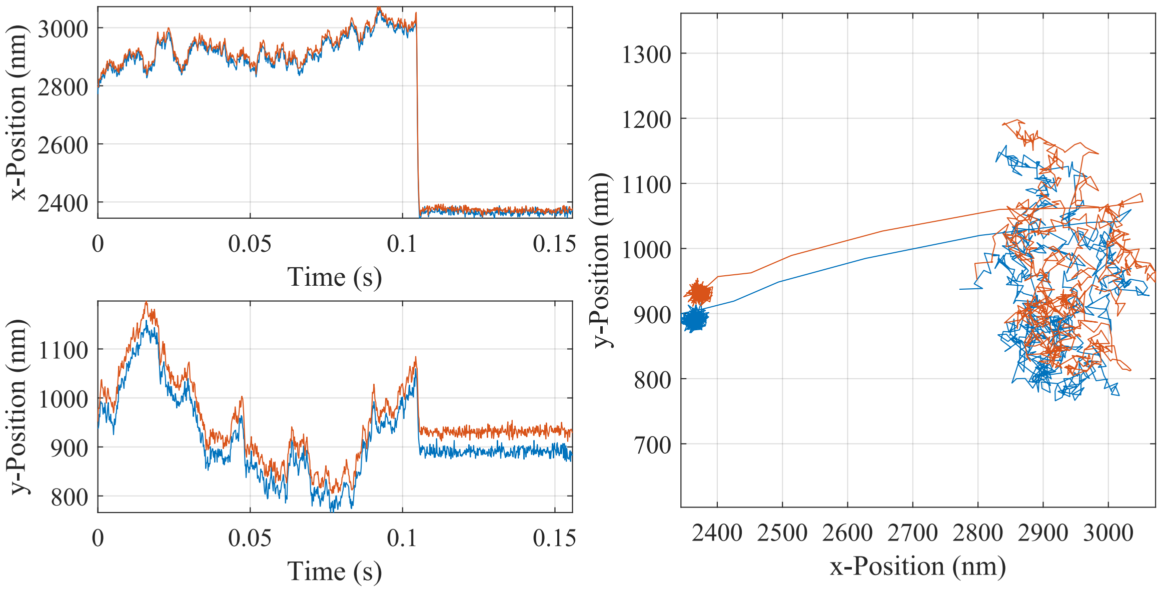

x-Position (nm)
2800
2700
2600
2500
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
800
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
800
2400
2500
2600
2700
2800
x-Position (nm)

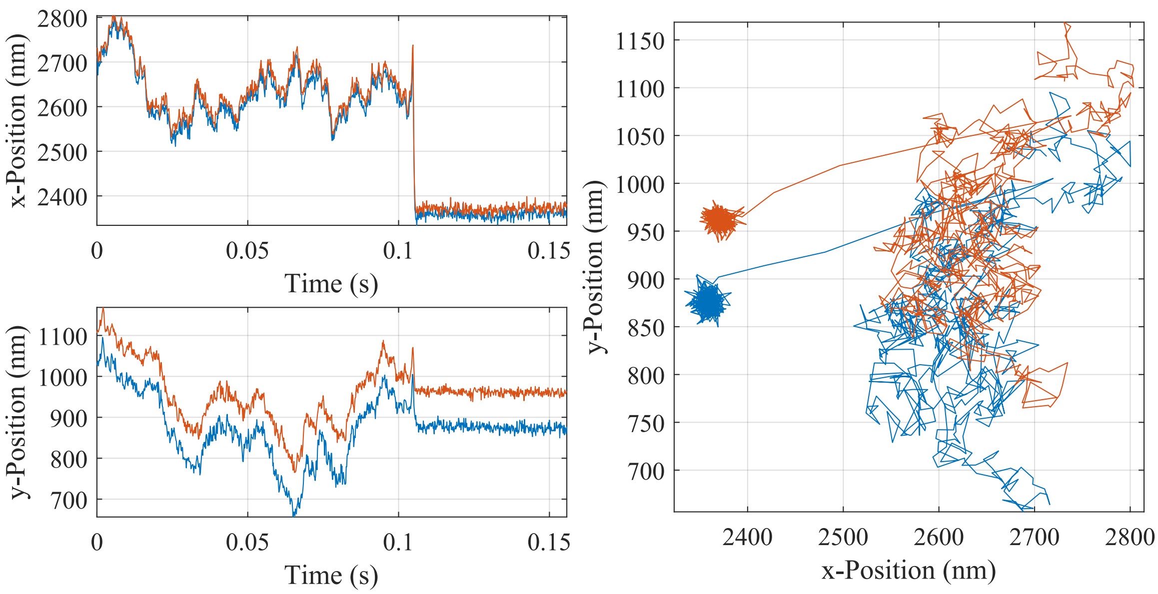

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

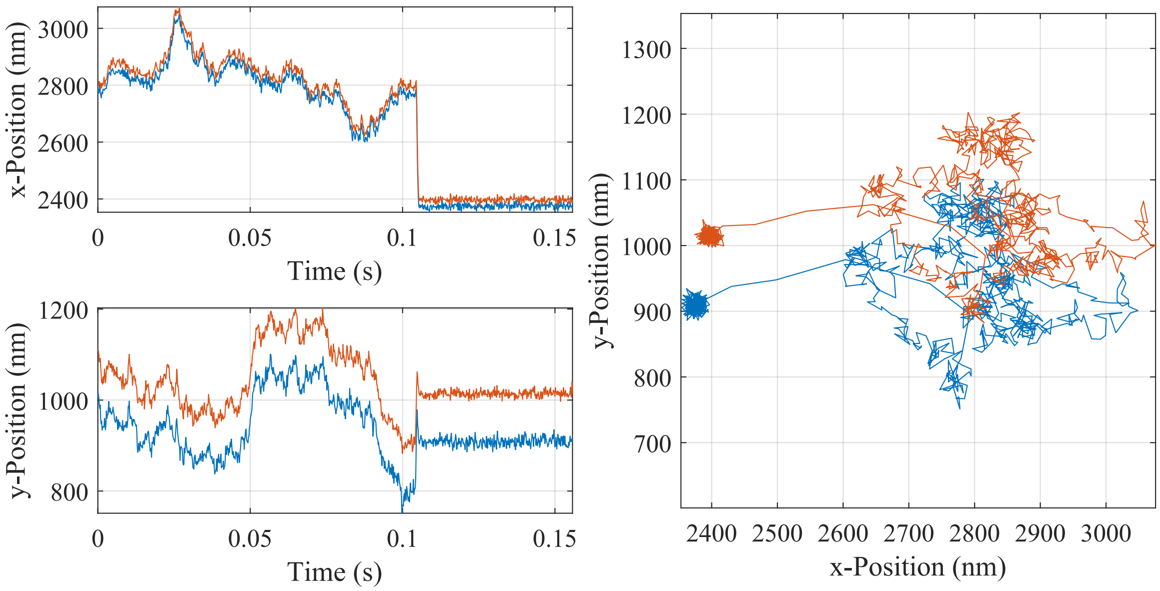

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

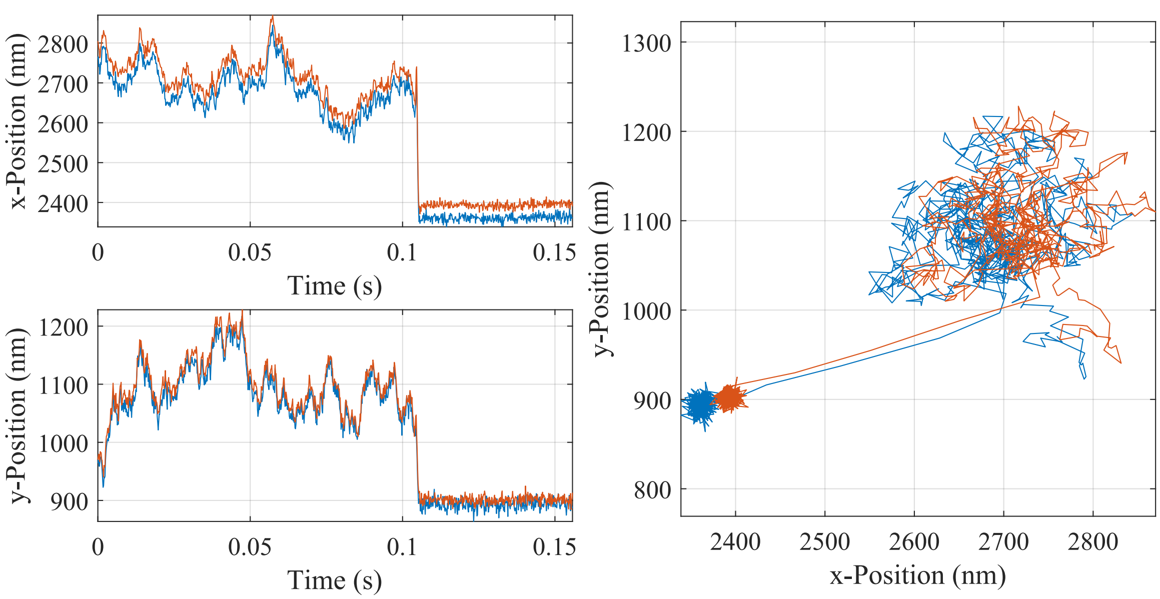

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1200
1000
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
900
2400
2500
2600
2700
2800
2900
x-Position (nm)

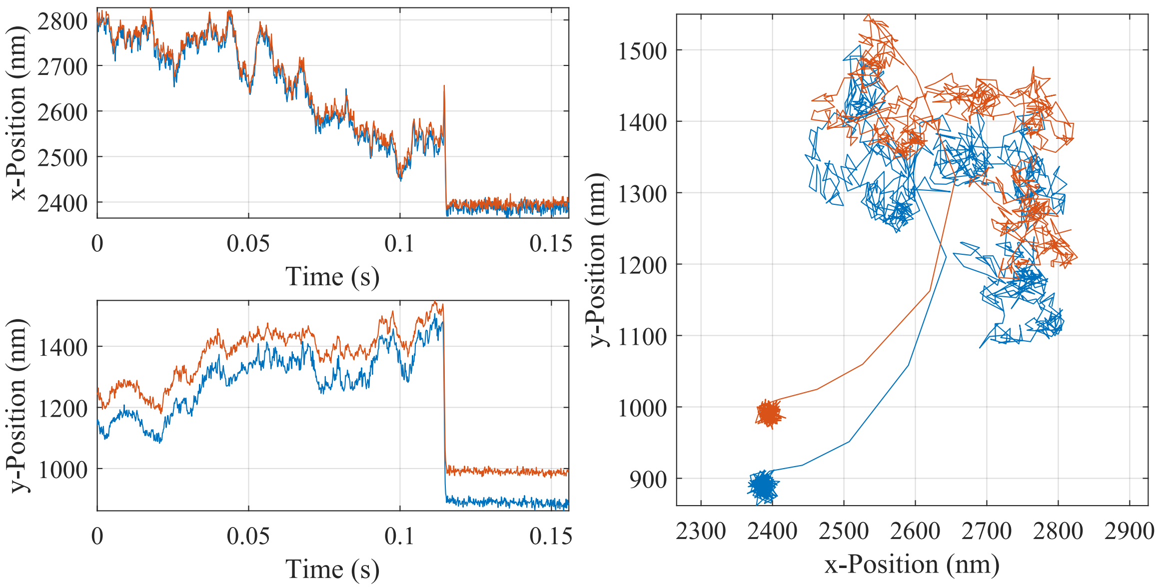

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
800
600
400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
900
800
700
600
500
400
2300
2400
2500
2600
2700
2800
x-Position (nm)

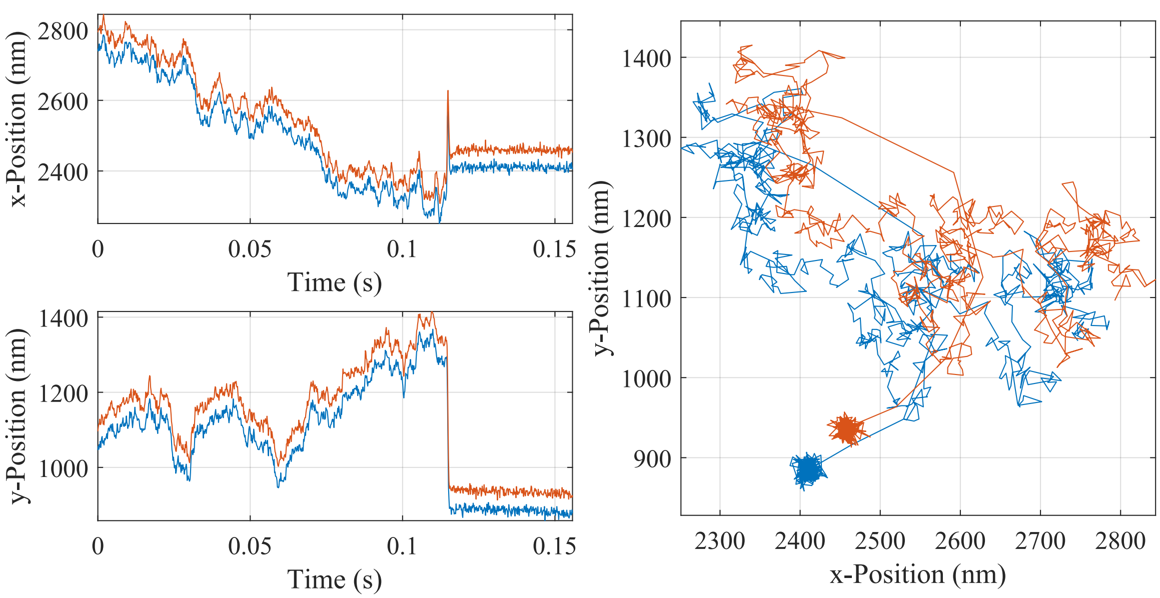

x-Position (nm)
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
800
700
2400 2500 2600 2700 2800 2900 3000
x-Position (nm)

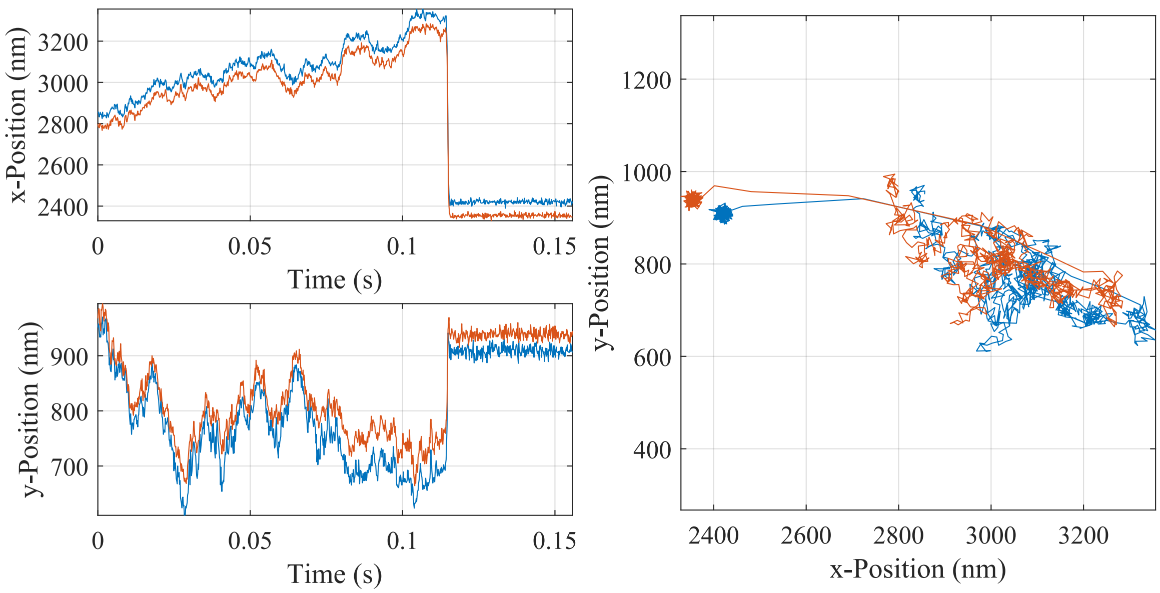

x-Position (nm)
2800
2700
2600
2500
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
700
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1150
1100
1050
1000
950
900
850
800
750
700
2400 2500 2600 2700 2800
x-Position (nm)

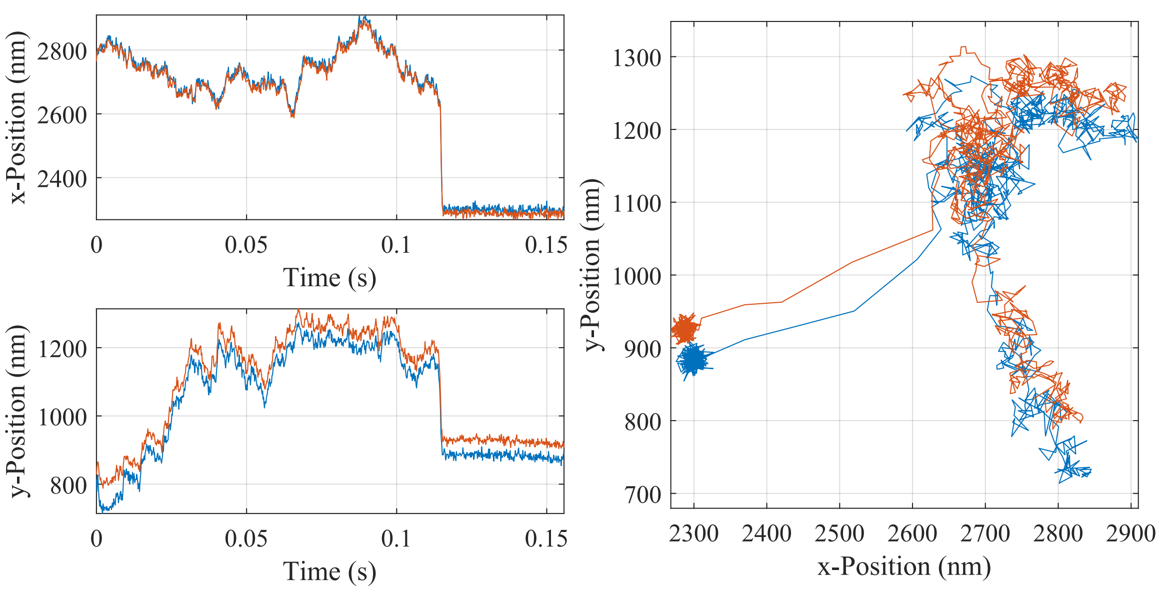

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

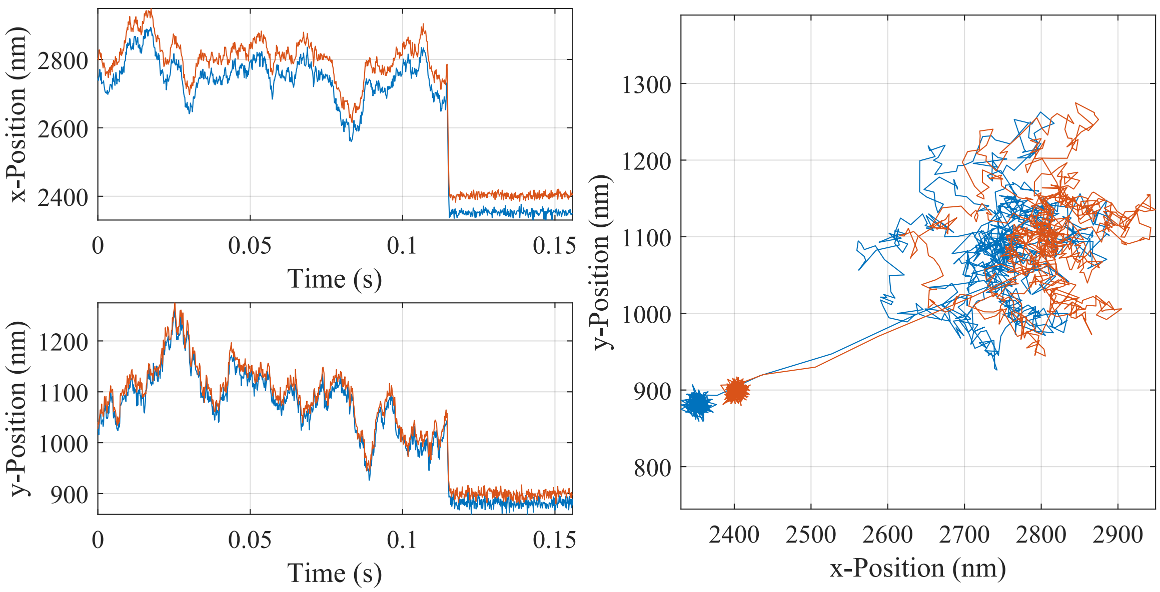

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

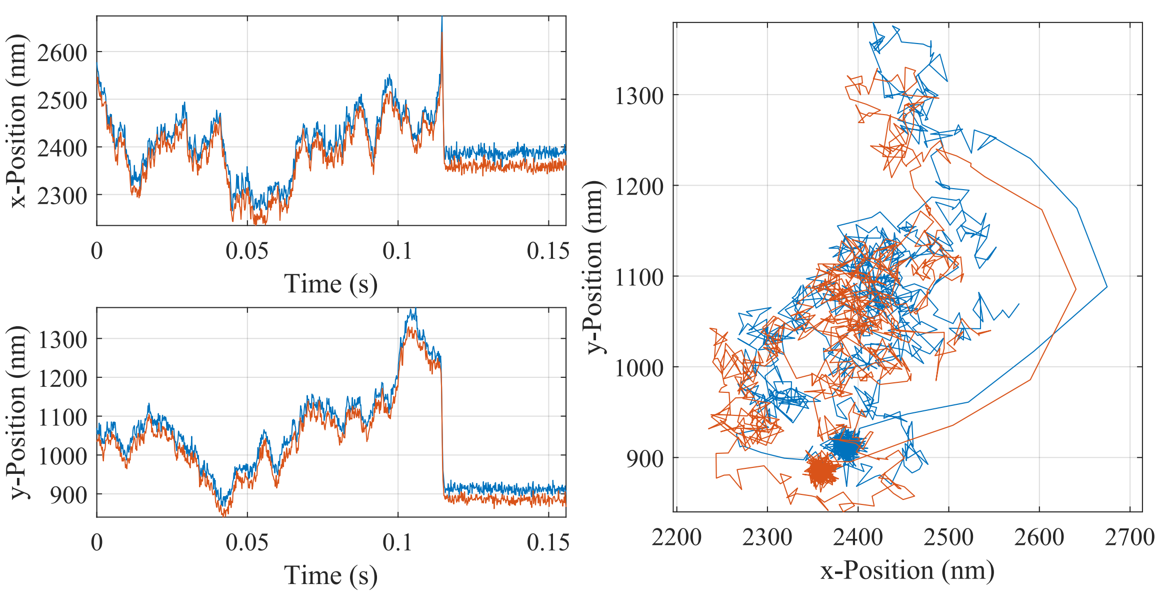

x-Position (nm)
2800
2700
2600
2500
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1200
1000
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
900
2300
2400
2500
2600
2700
2800
2900
x-Position (nm)

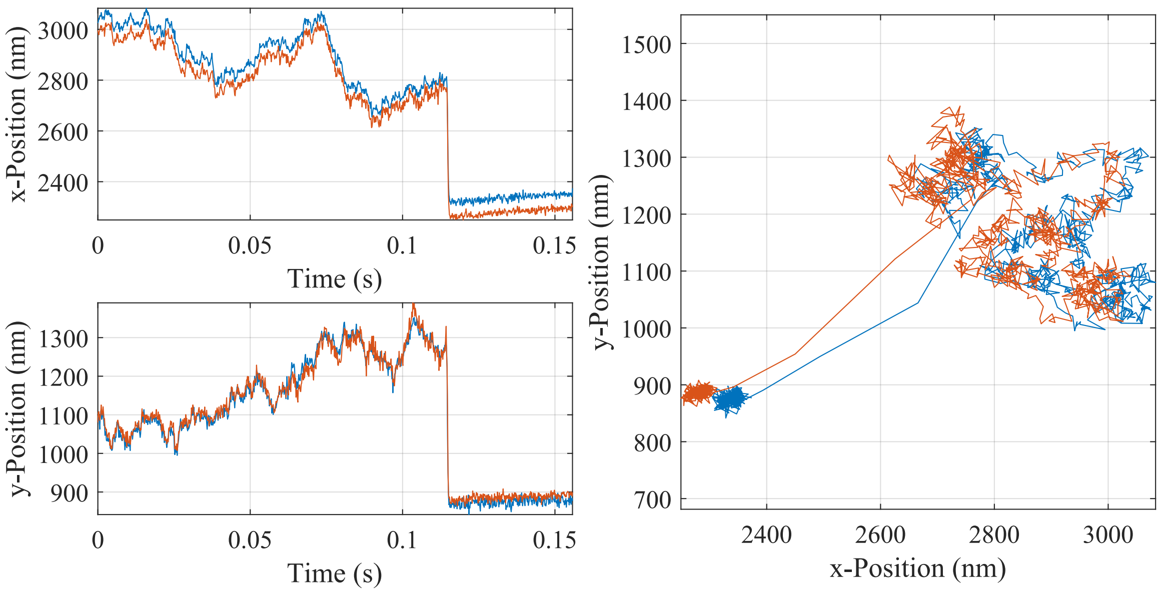

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1200
1000
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1300
1200
1100
1000
900
2300
2400
2500
2600
2700
2800
x-Position (nm)

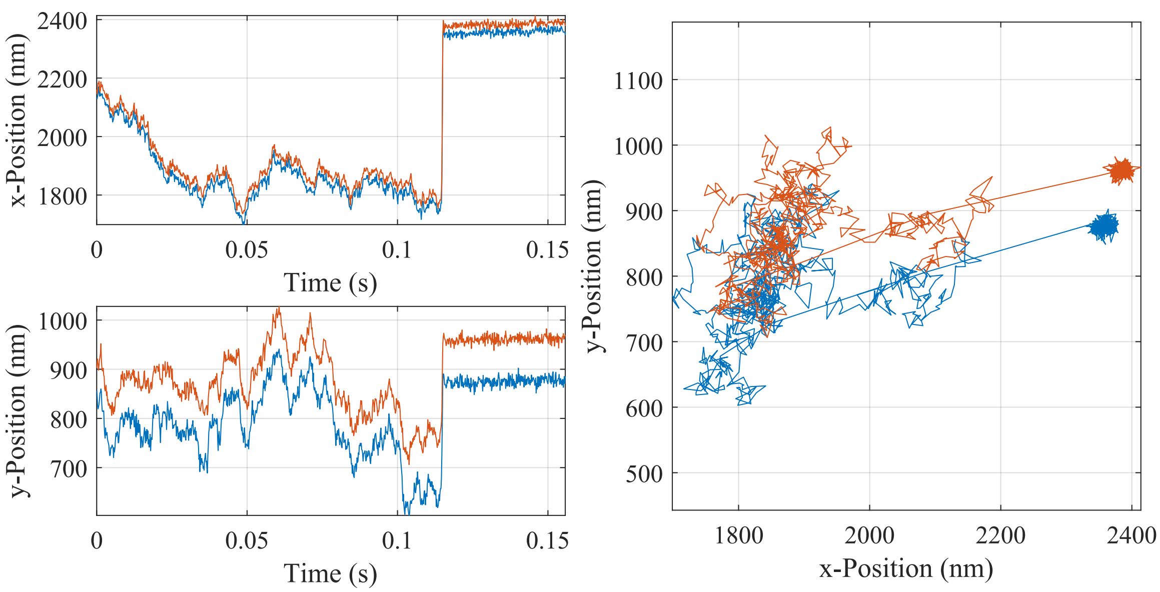

x-Position (nm)
3200
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
900
800
700
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1000
800
600
400
2400
2600
2800
3000
3200
x-Position (nm)

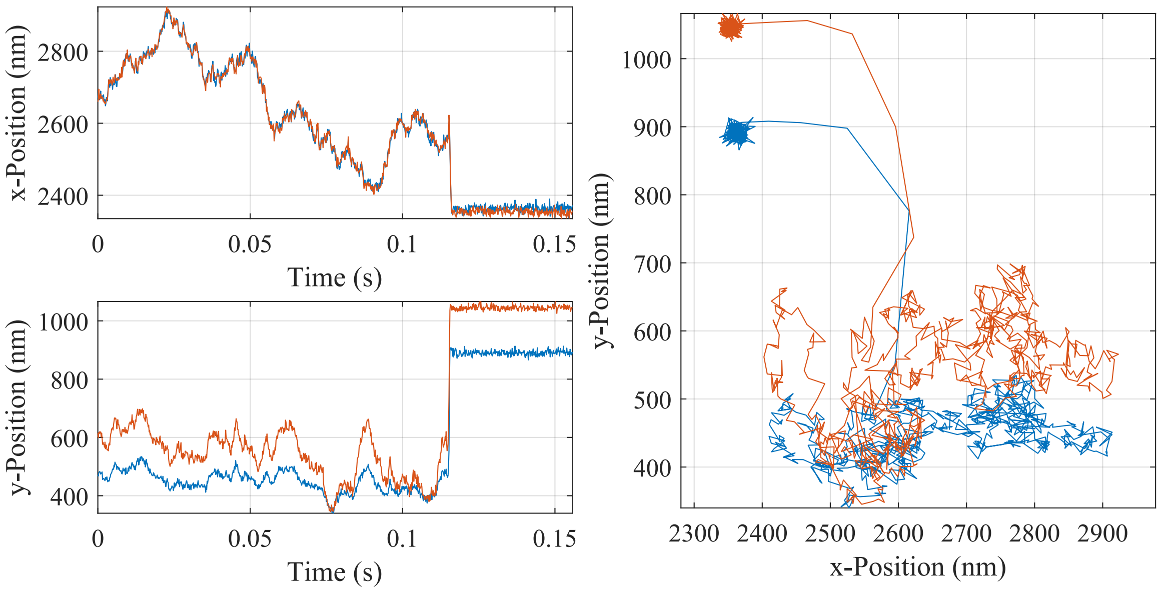

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1000
800
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
800
700
2300
2400
2500
2600
2700
2800
2900
x-Position (nm)

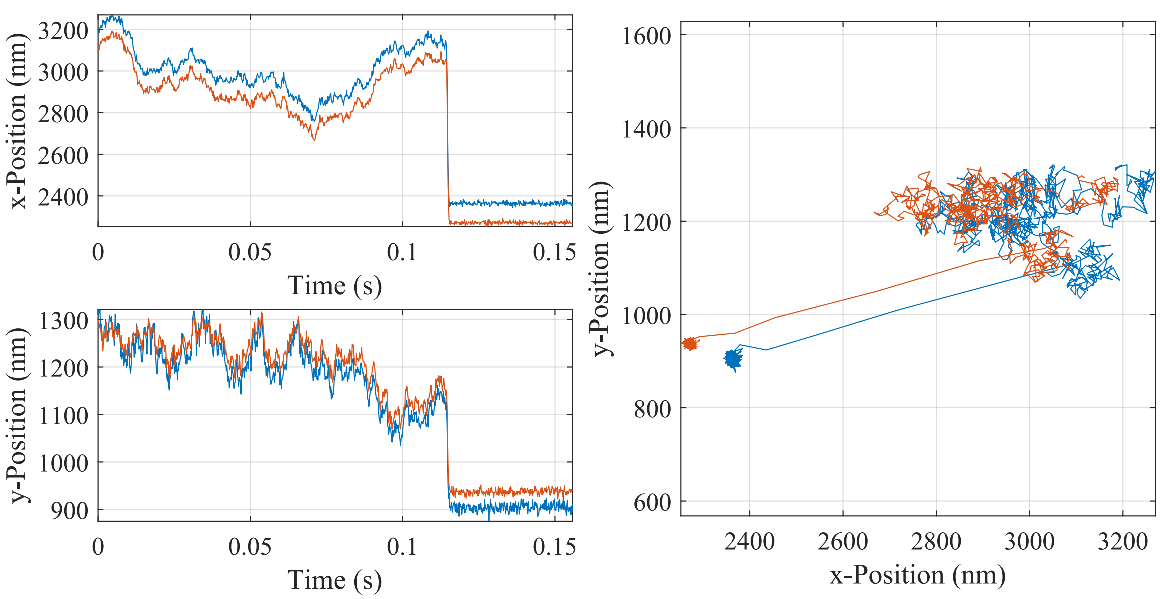

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
800
2400
2500
2600
2700
2800
2900
x-Position (nm)

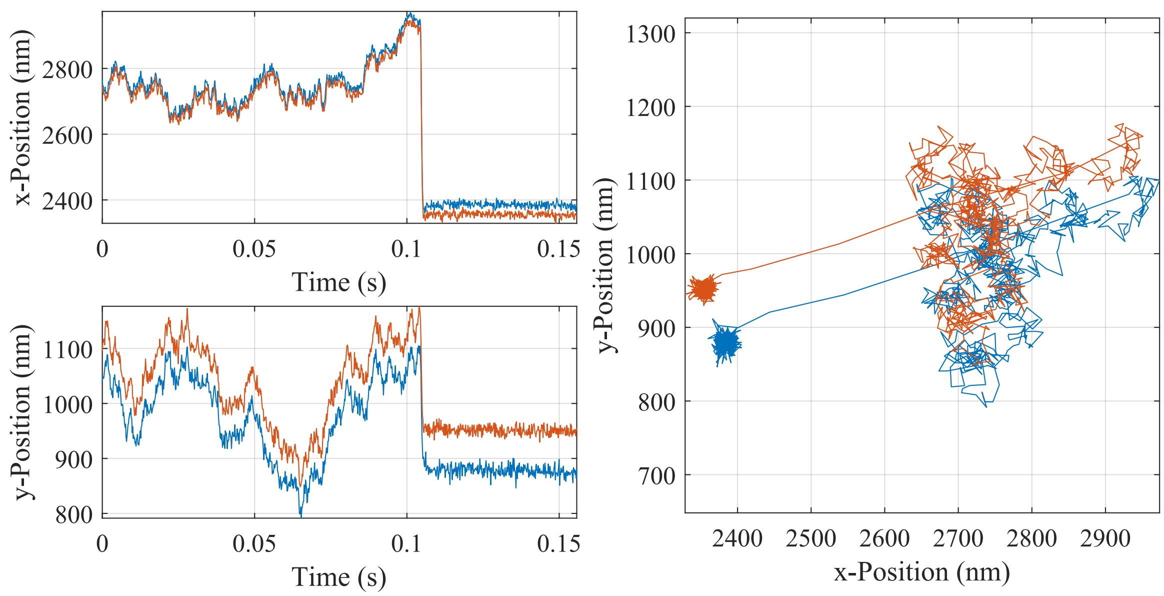

x-Position (nm)
2600
2500
2400
2300
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
2200
2300
2400
2500
2600
2700
x-Position (nm)

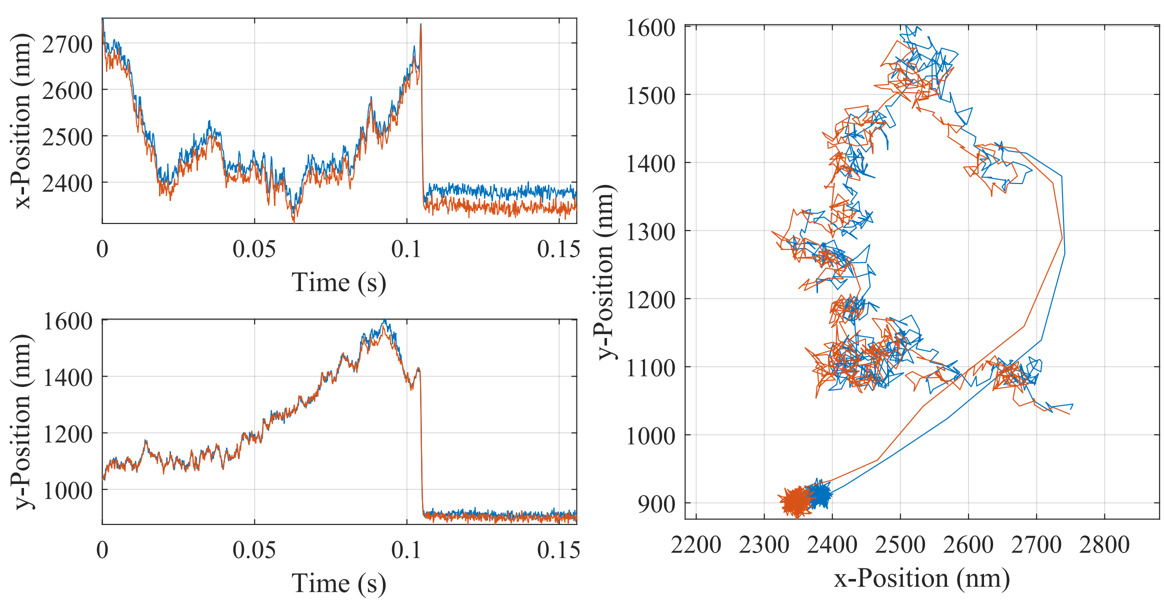

x-Position (nm)
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
900
800
700
2400
2600
2800
3000
x-Position (nm)

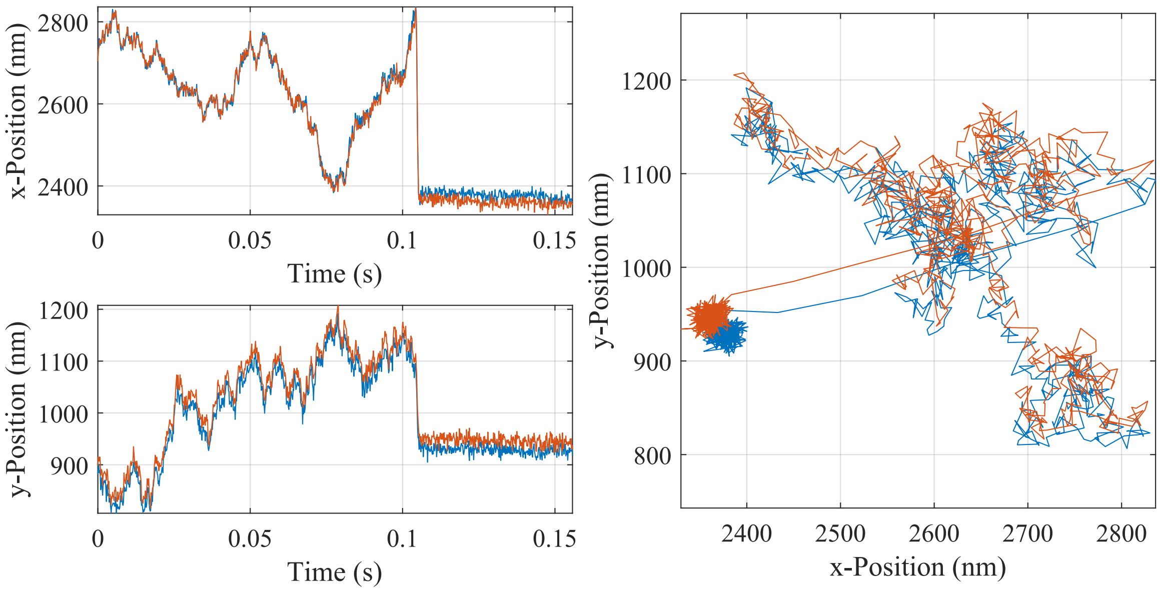

x-Position (nm)
2400
2200
2000
1800
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
900
800
700
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
700
600
500
1800
2000
2200
2400
x-Position (nm)

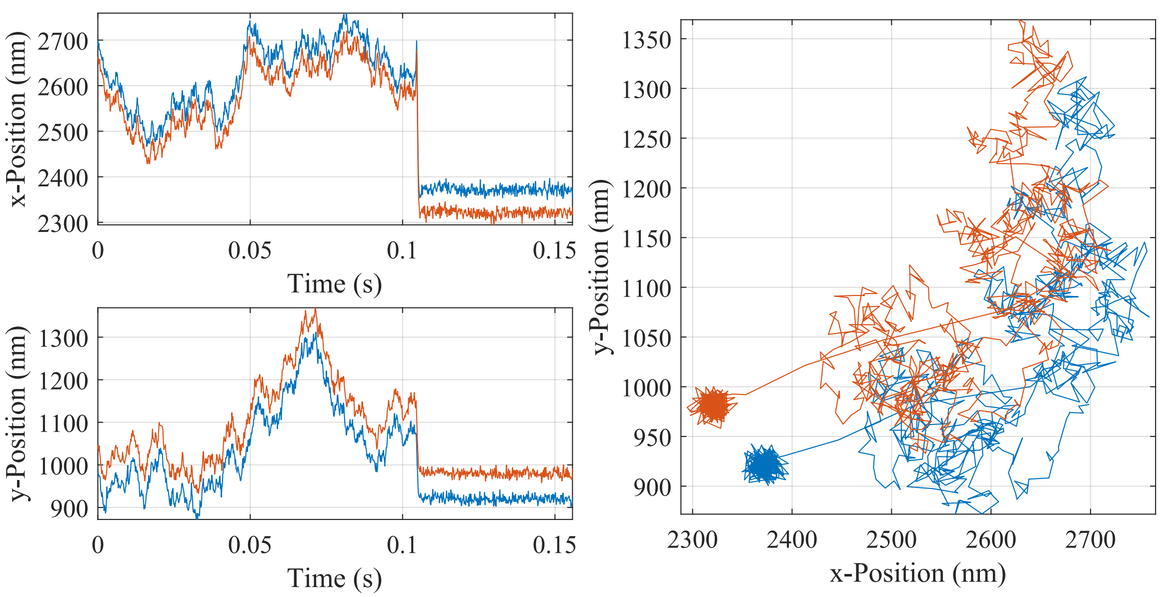

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
800
600
400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
900
800
700
600
500
400
2300
2400
2500
2600
2700
2800
2900
x-Position (nm)

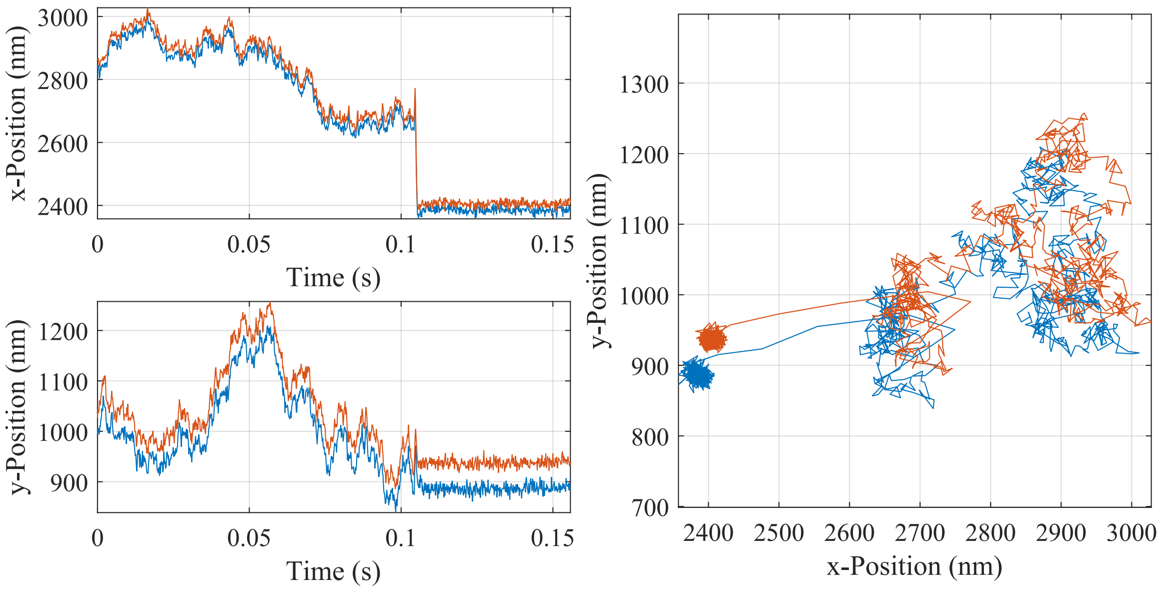

x-Position (nm)
3200
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1600
1400
1200
1000
800
600
2400
2600
2800
3000
3200
x-Position (nm)

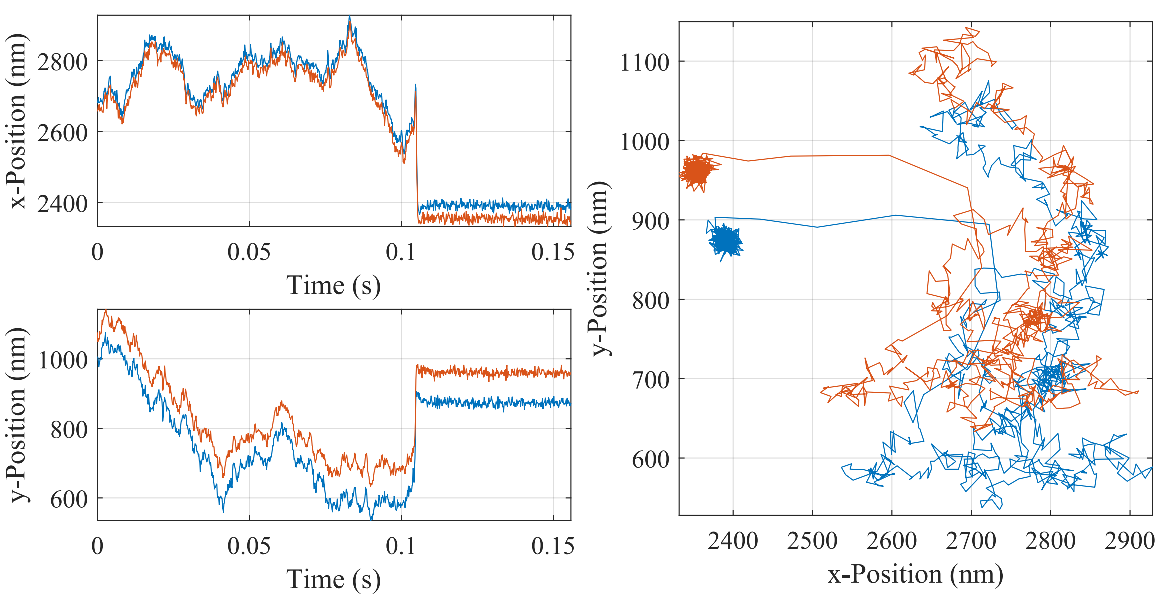

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

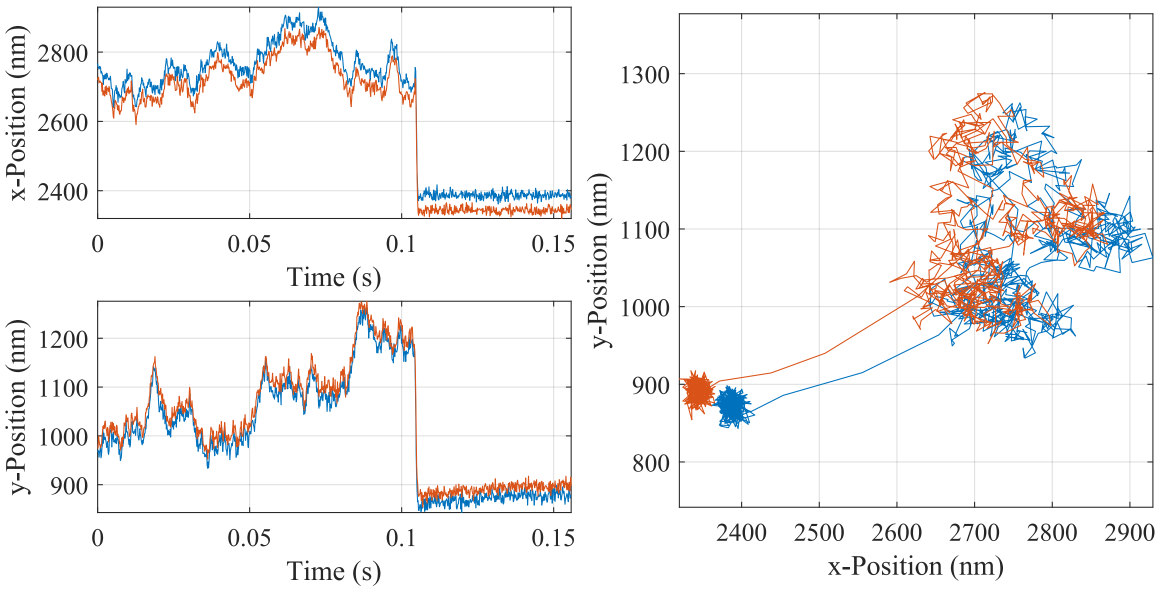

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

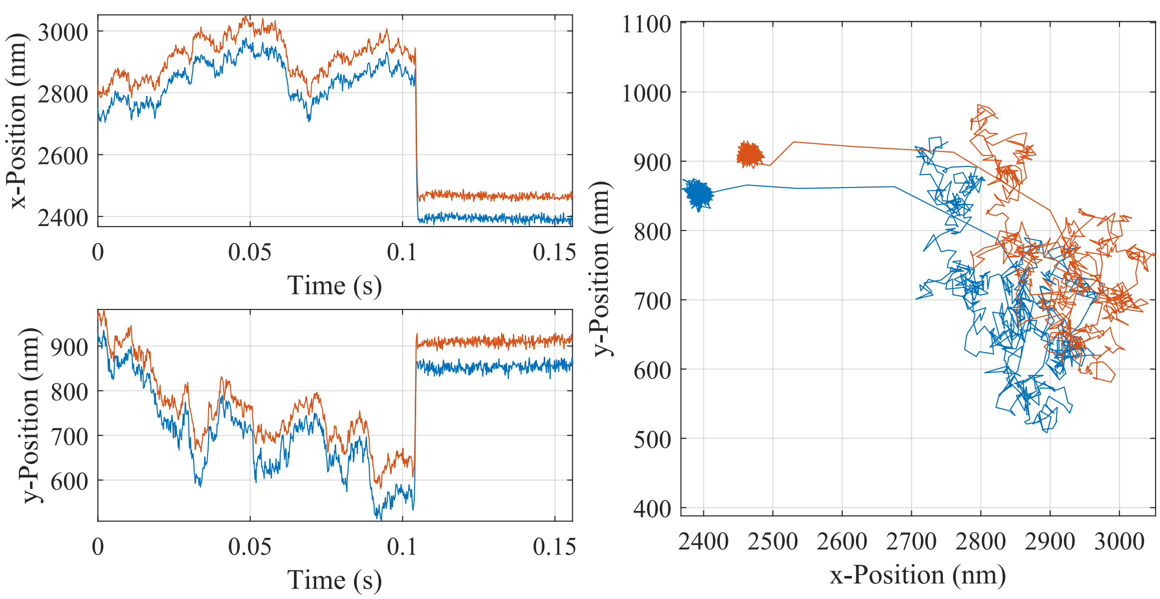

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
800
2400
2500
2600
2700
2800
x-Position (nm)

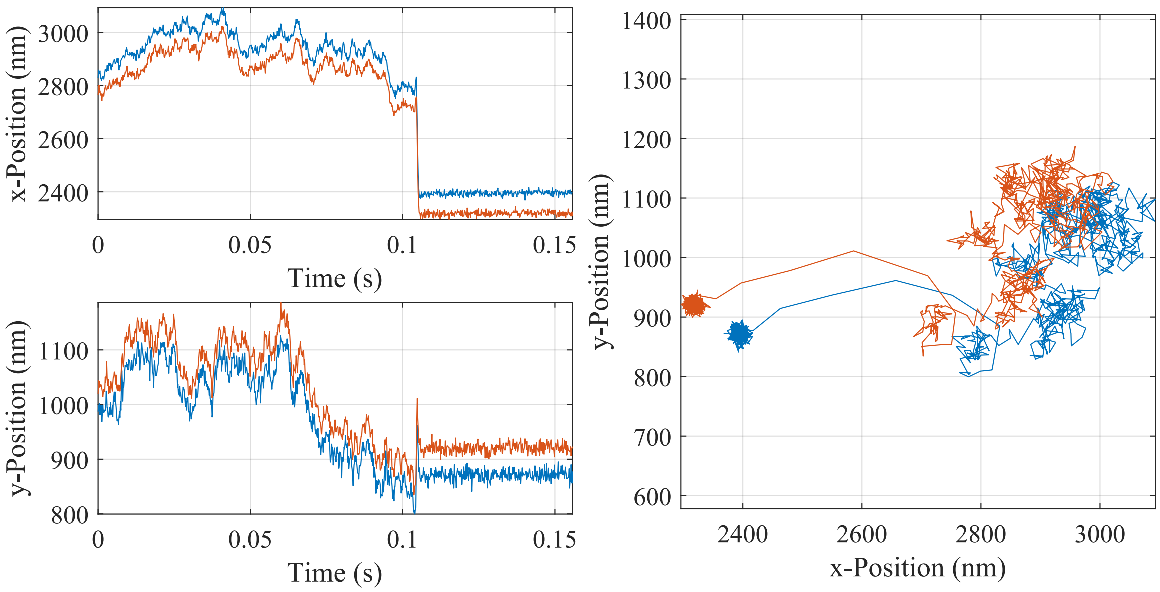

x-Position (nm)
2700
2600
2500
2400
2300
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1350
1300
1250
1200
1150
1100
1050
1000
950
900
2300
2400
2500
2600
2700
x-Position (nm)

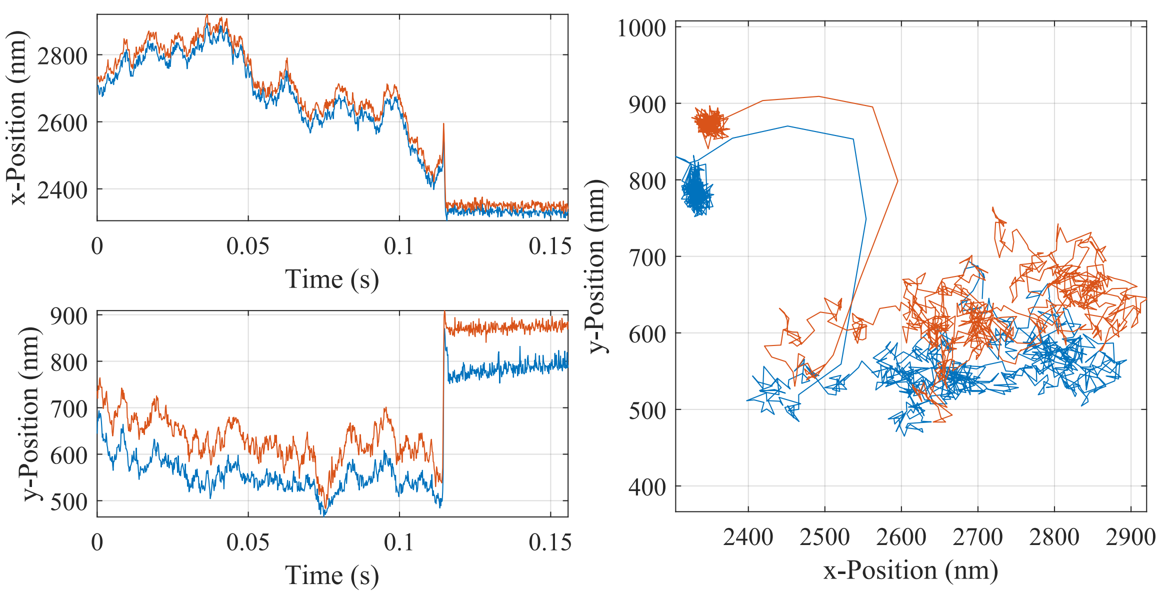

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

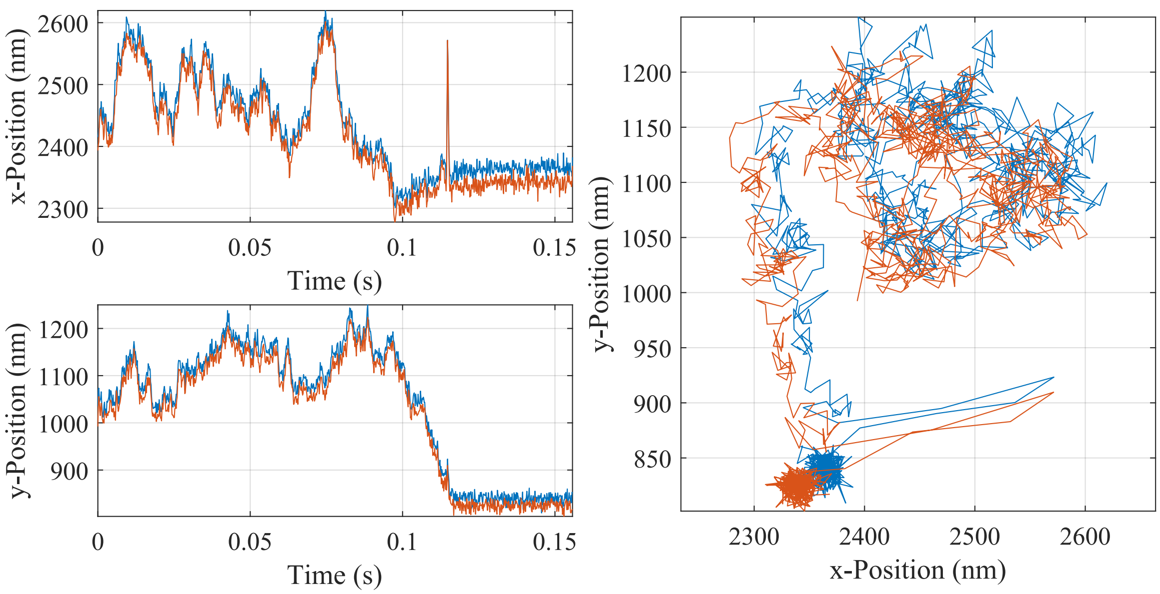

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)


x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1200
1100
1000
900
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1300
1200
1100
1000
900
800
2400
2500
2600
2700
2800
2900
x-Position (nm)


x-Position (nm)
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
900
800
700
600
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
700
600
500
400
2400
2500
2600
2700
2800
2900
3000
x-Position (nm)

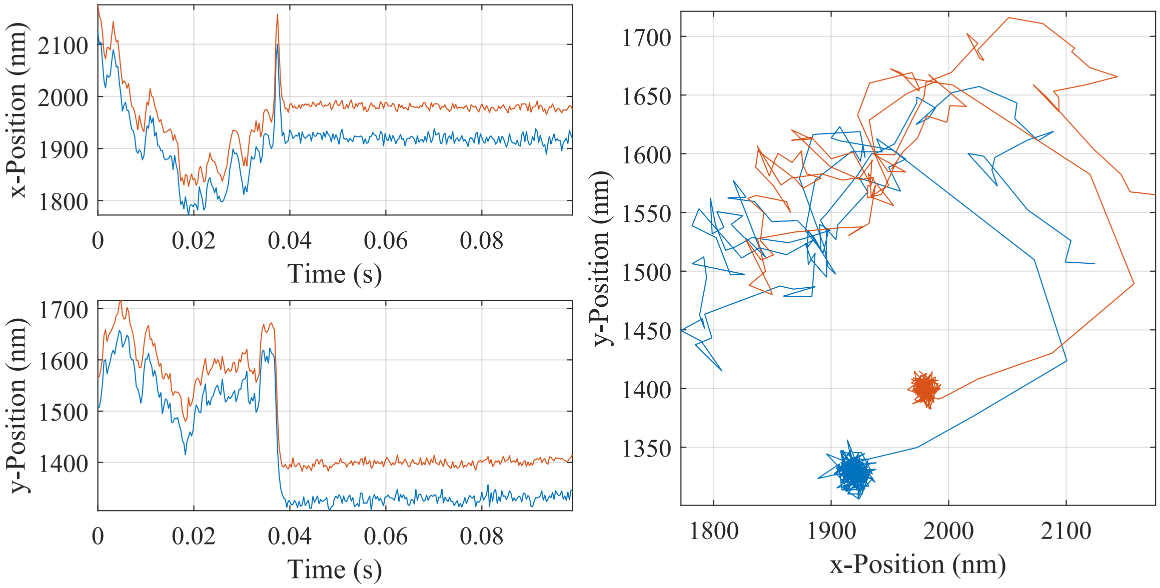

x-Position (nm)
3000
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1100
1000
900
800
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1400
1300
1200
1100
1000
900
800
700
600
2400
2600
2800
3000
x-Position (nm)

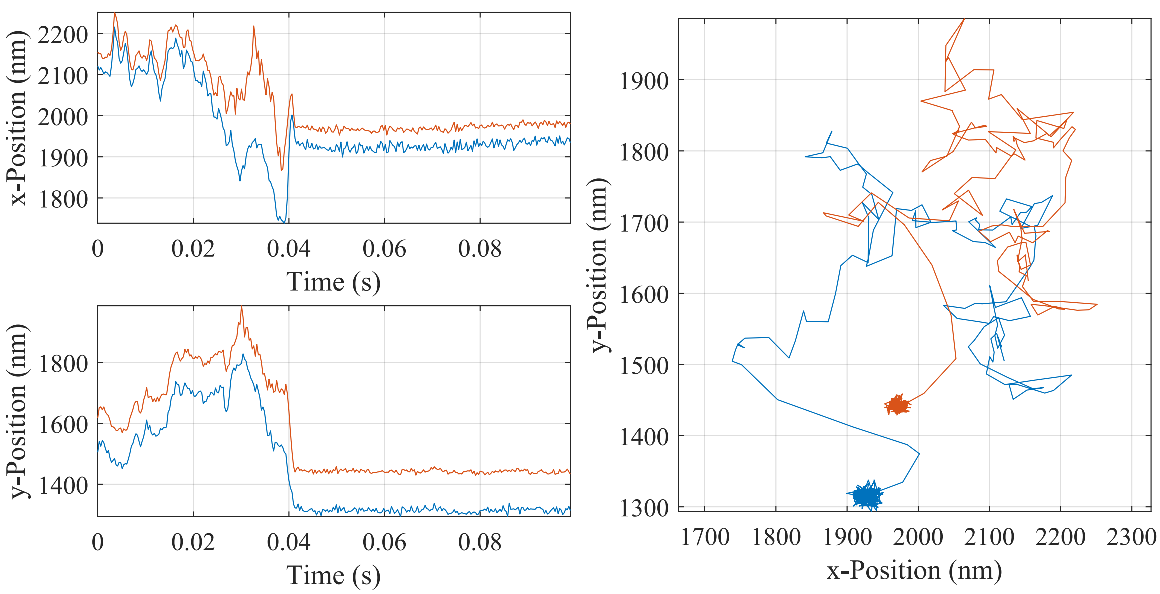

x-Position (nm)
2800
2600
2400
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
900
800
700
600
500
0
0.05
0.1
0.15
Time (s)
y-Position (nm)
1000
900
800
700
600
500
400
2400
2500
2600
2700
2800
2900
x-Position (nm)

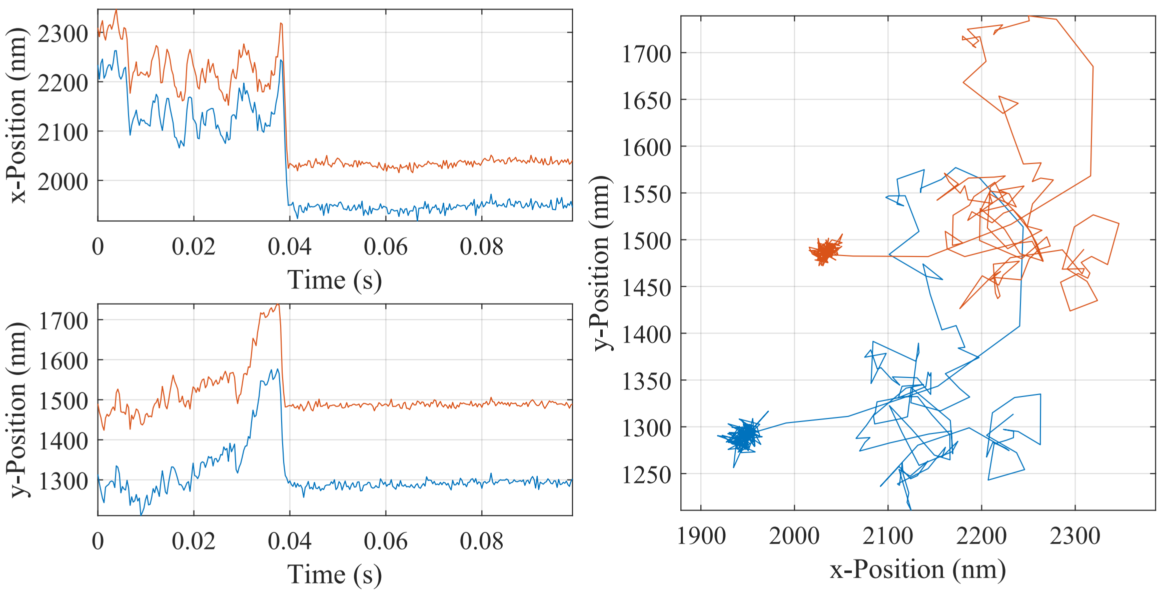

2600
2500
2400
2300
x-Position (nm)
0
0.05
0.1
0.15
Time (s)
1200
1100
1000
900
y-Position (nm)
0
0.05
0.1
0.15
Time (s)
1200
1150
1100
1050
1000
950
900
850
y-Position (nm)
2300
2400
2500
2600
x-Position (nm)

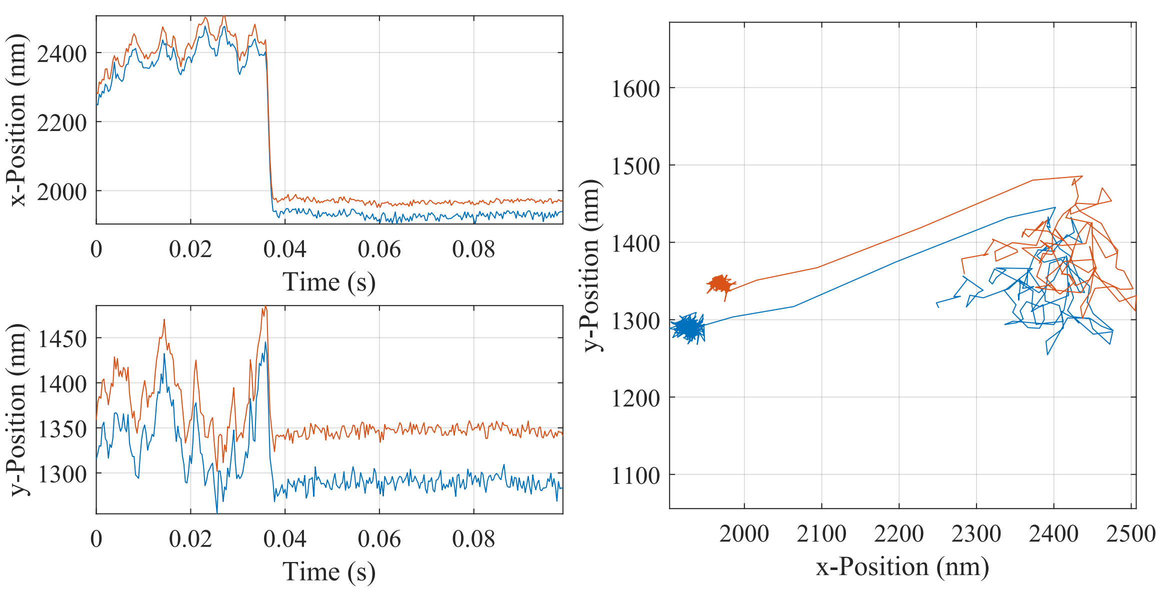
APPENDIX E

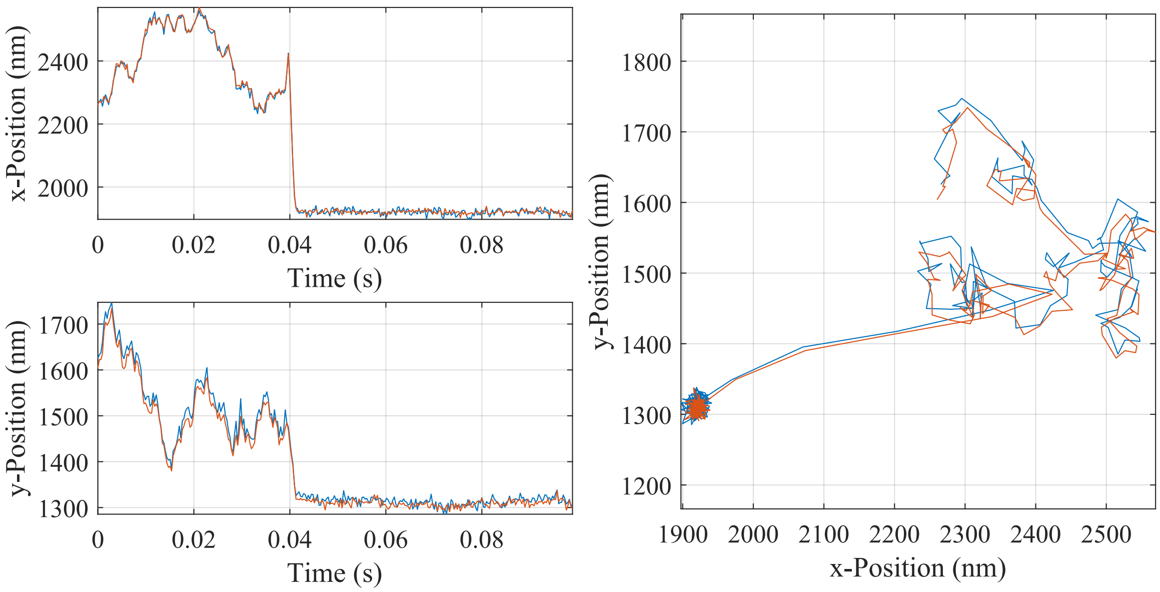
TRR Experiments Data for 500nm Diameter Particle

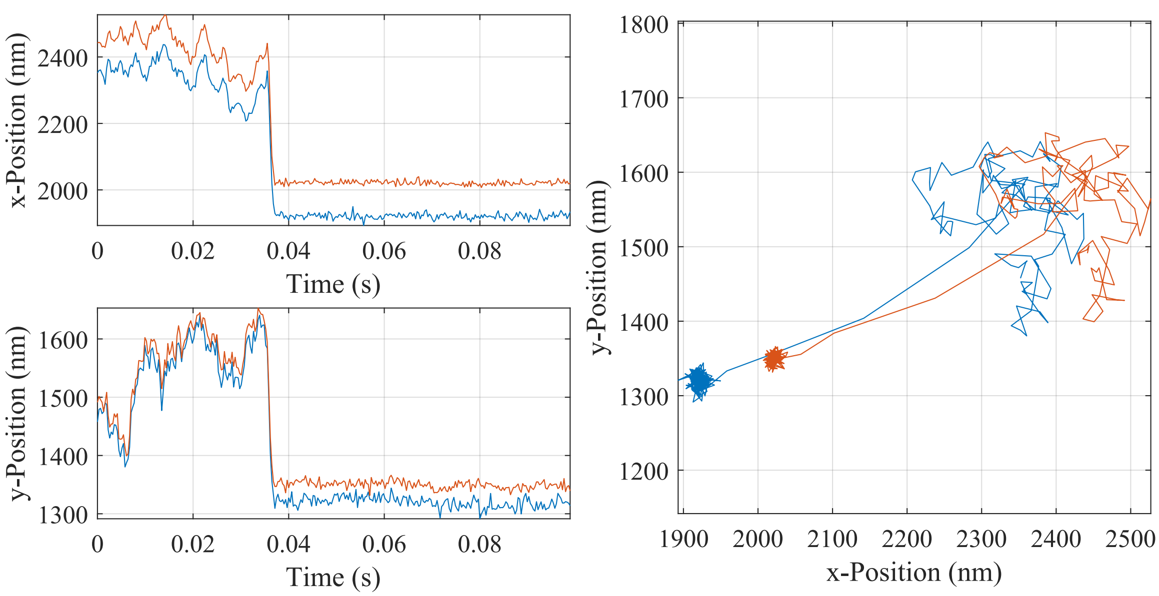

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

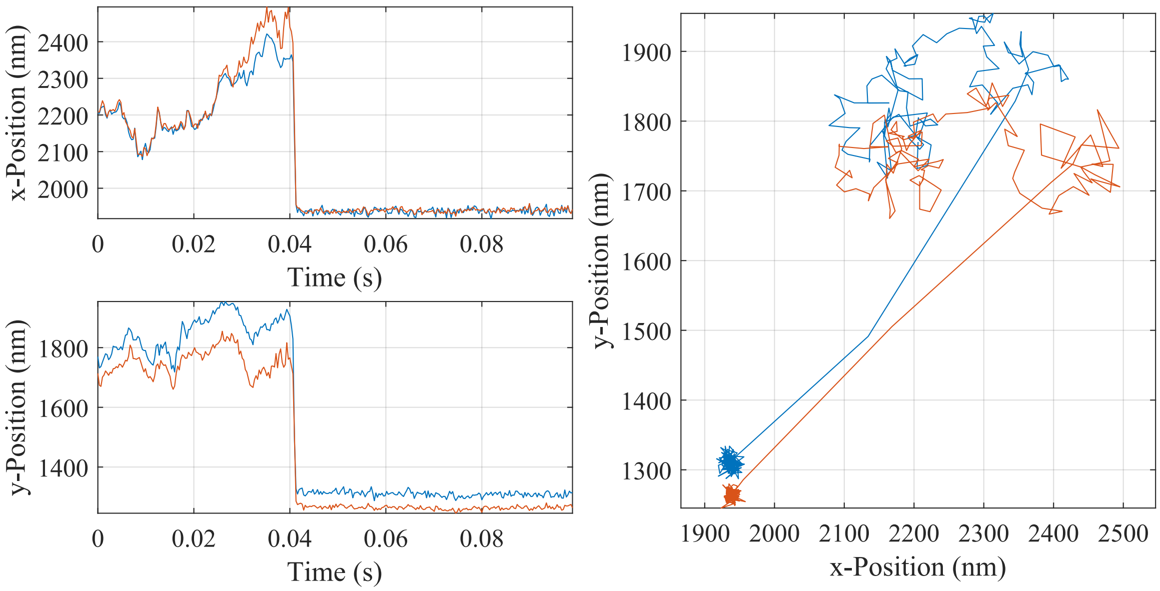

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

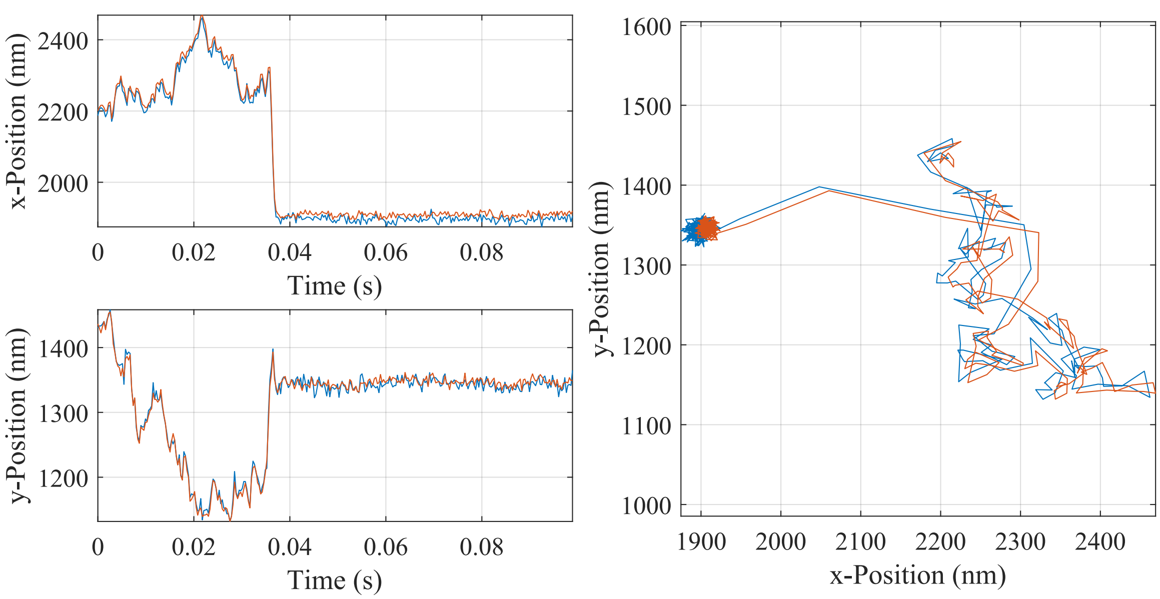

x-Position (nm)
2300
2200
2100
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1700
1600
1500
1400
1300
Time (s)
y-Position (nm)
1650
1550
1450
1350
1250
1900
2000
2100
2200
2300
x-Position (nm)

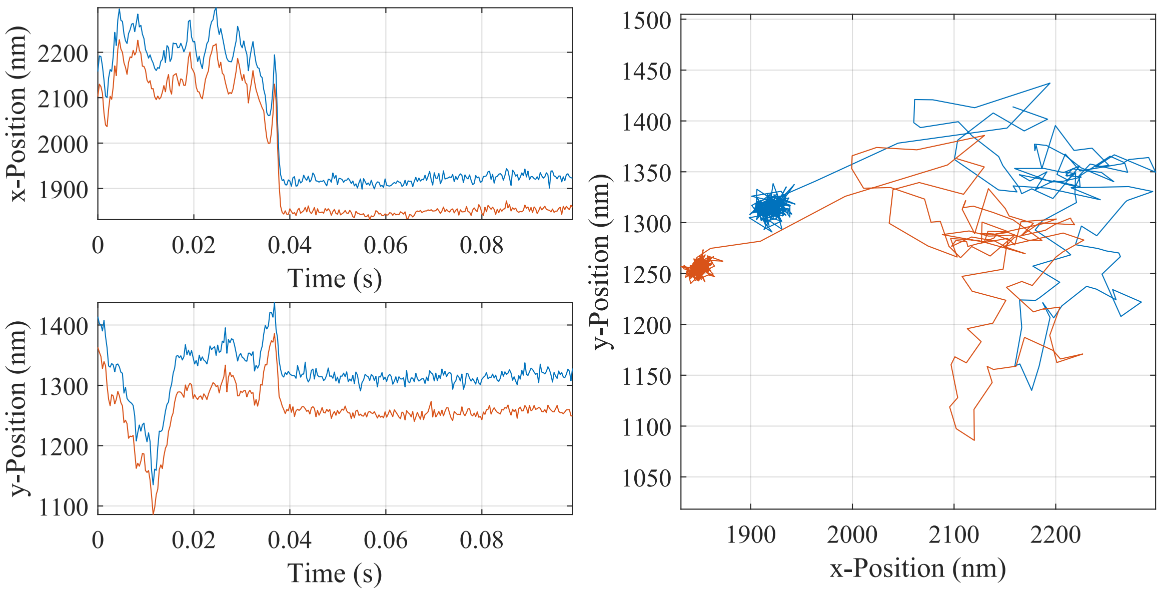

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1450
1400
1350
1300
Time (s)
y-Position (nm)
1600
1500
1400
1300
1200
1100
2000
2100
2200
2300
2400
2500
x-Position (nm)

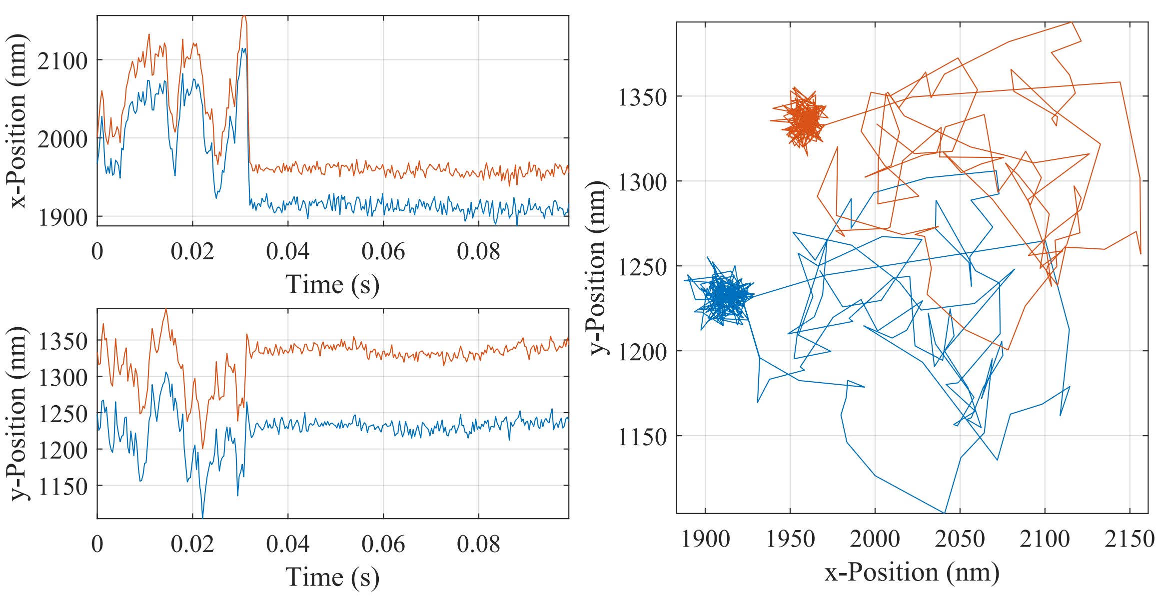

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1700
1600
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
1800
1700
1600
1500
1400
1300
1200
y-Position (nm)
1900
2000
2100
2200
2300
2400
2500
x-Position (nm)

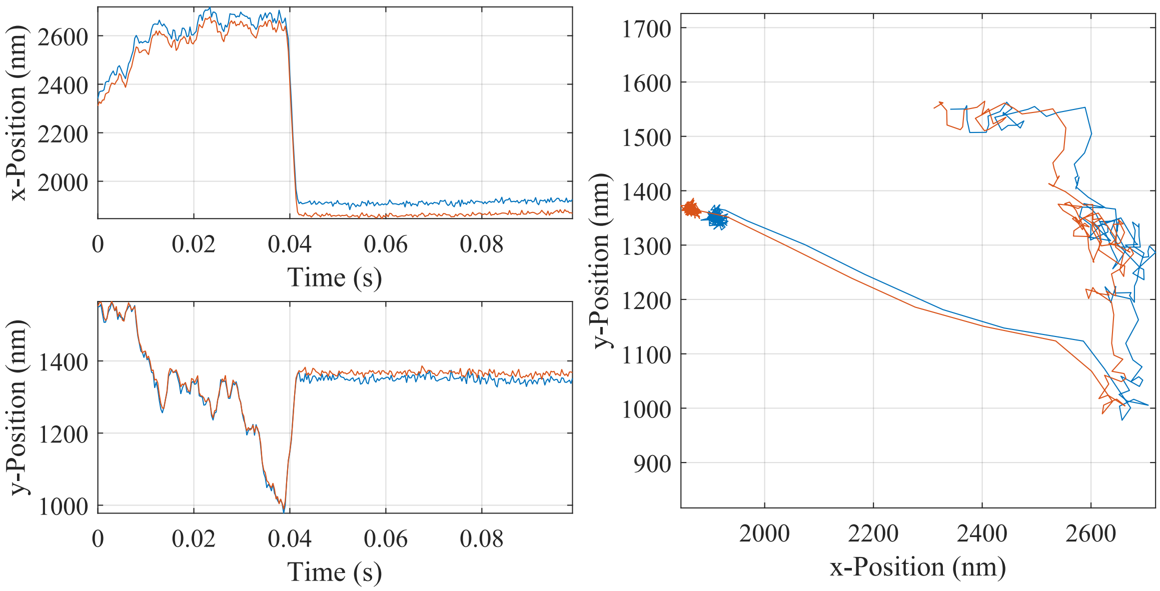

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1600
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
1800
1700
1600
1500
1400
1300
1200
y-Position (nm)
1900
2000
2100
2200
2300
2400
2500
x-Position (nm)

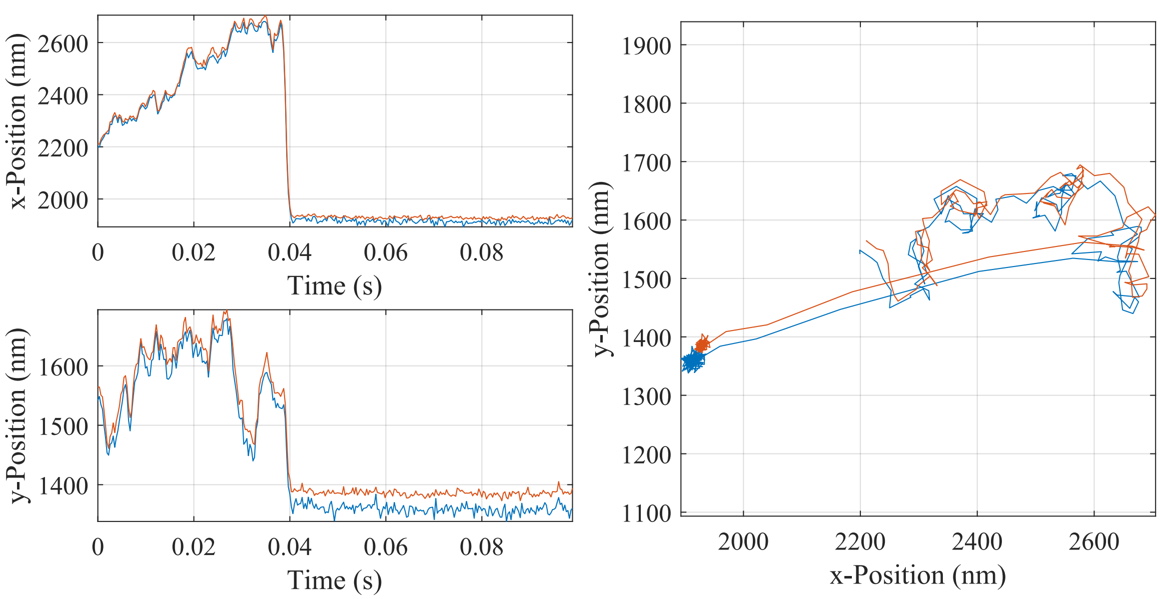

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

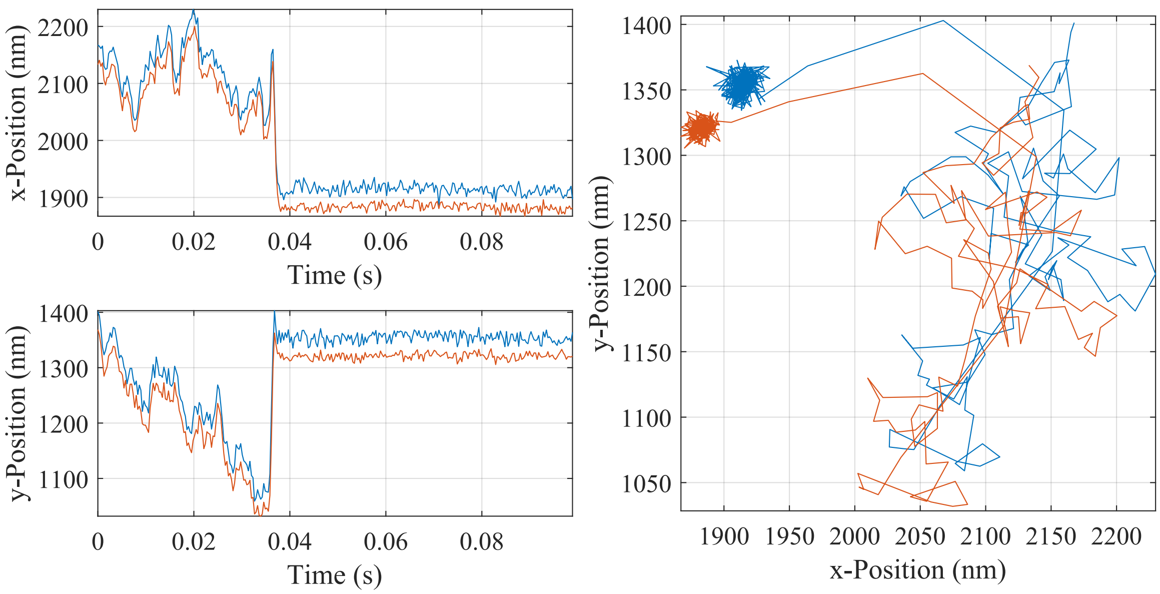

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

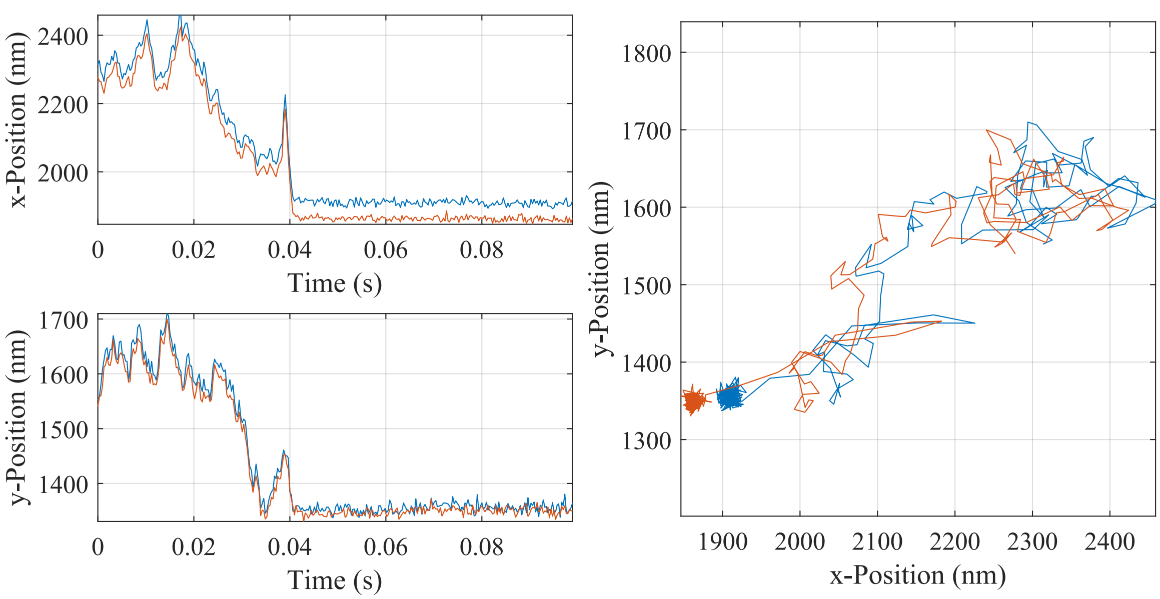

x-Position (nm)
2200
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1400
1300
1200
1100
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1450
1400
1350
1300
1250
1200
1150
1100
1050
1900
2000
2100
2200
x-Position (nm)

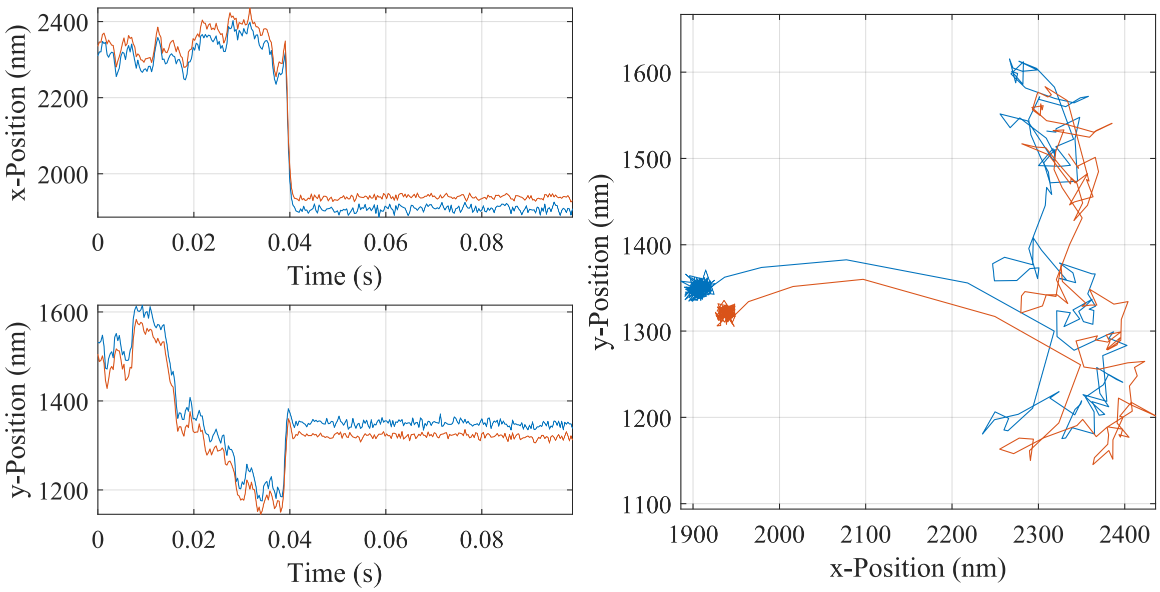

x-Position (nm)
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1350
1300
1250
1200
1150
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1350
1300
1250
1200
1150
1900
1950
2000
2050
2100
2150
x-Position (nm)

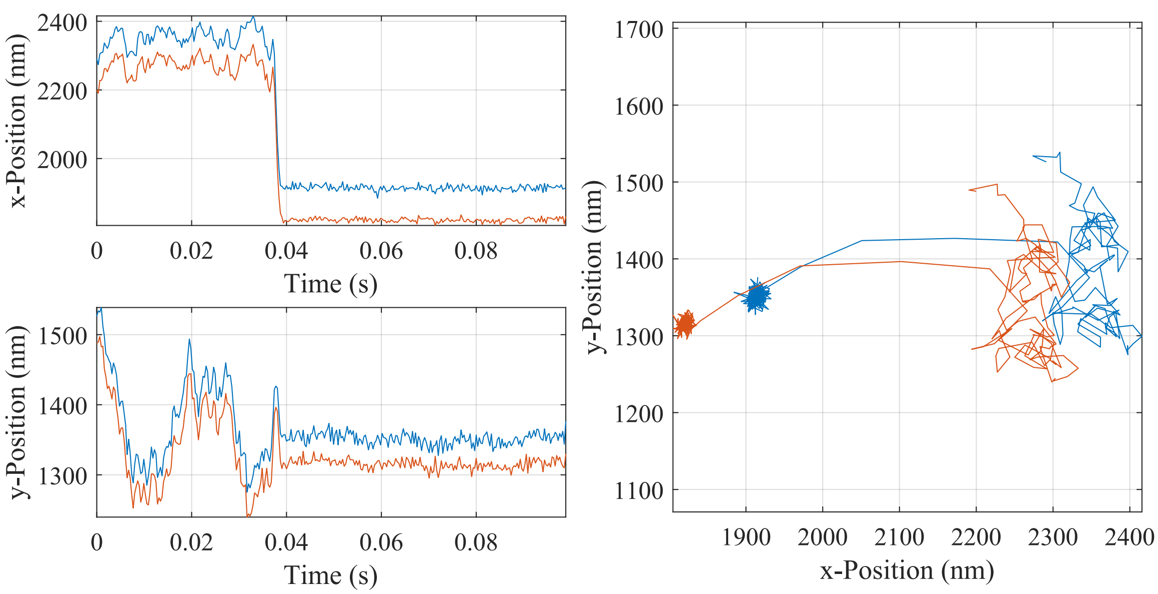

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

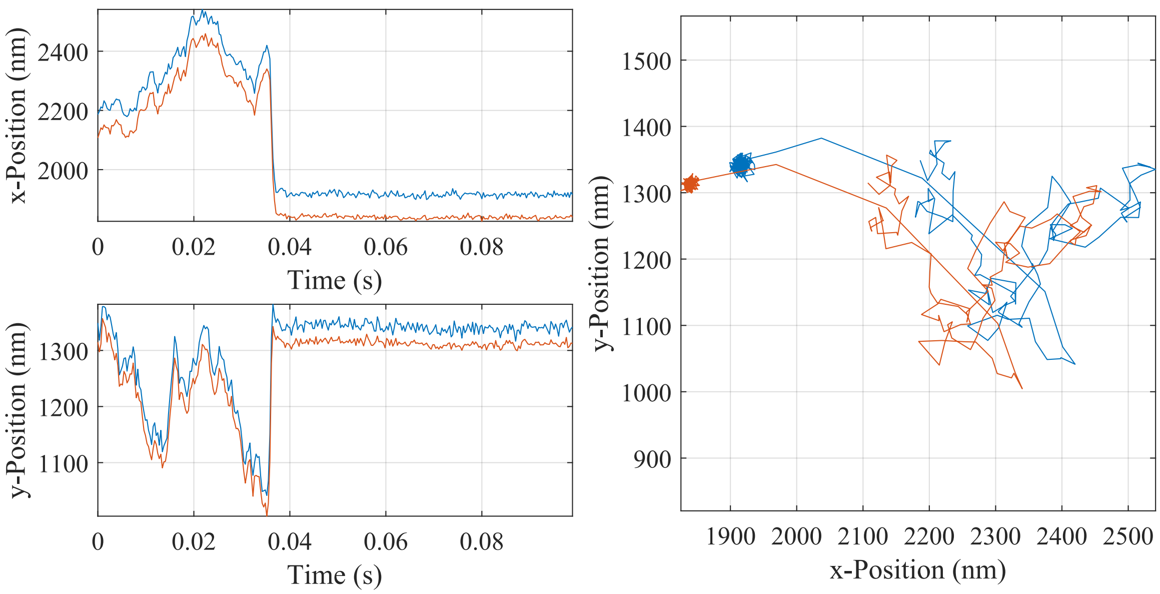

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

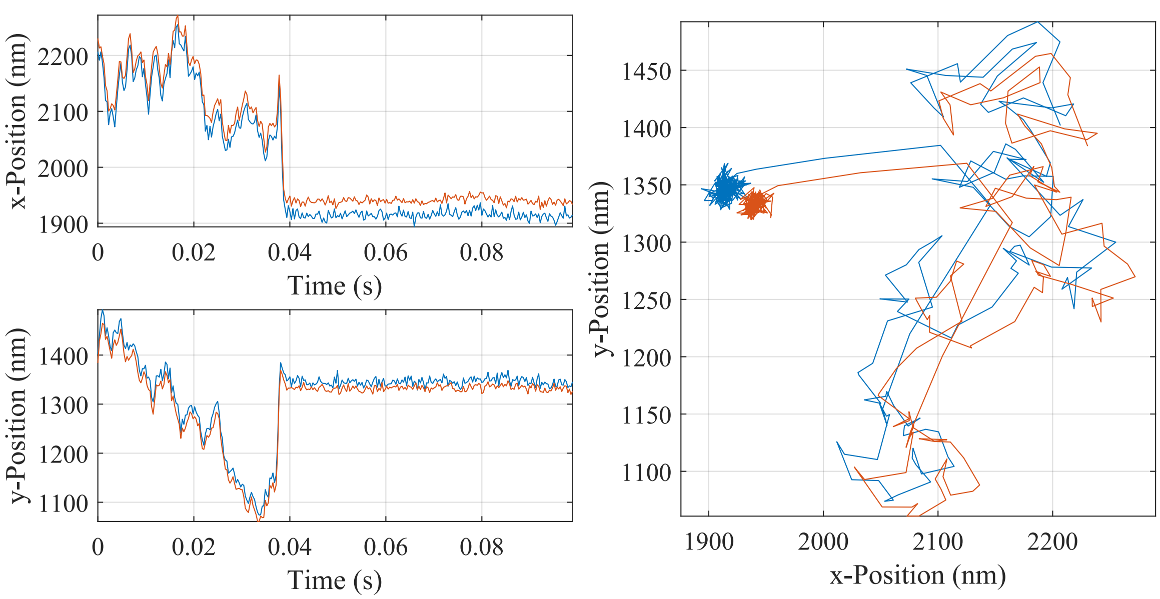

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

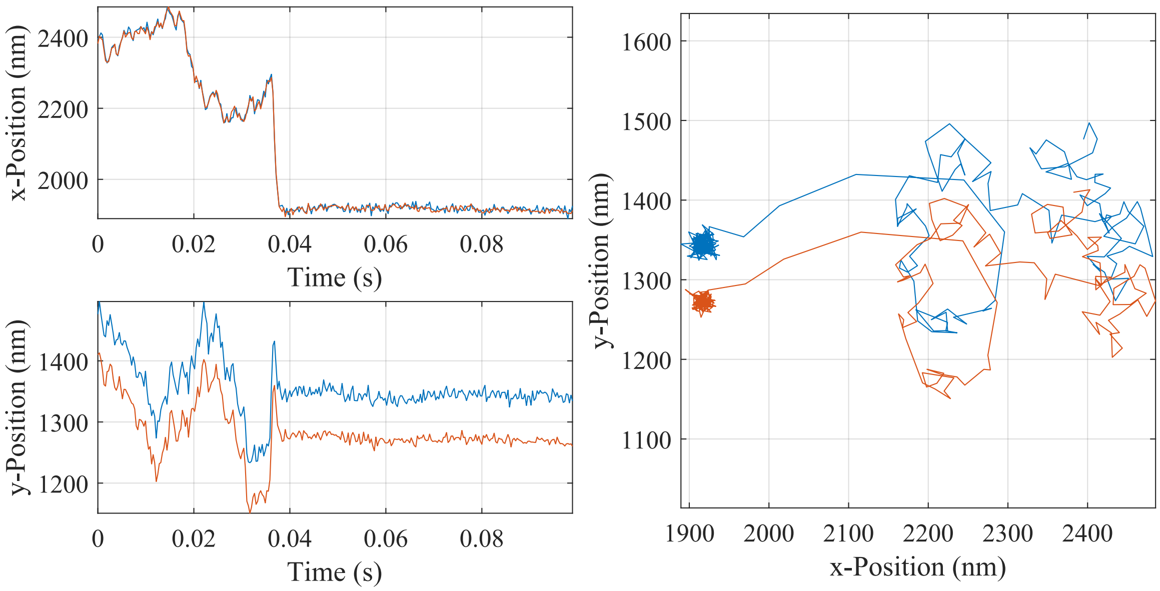

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

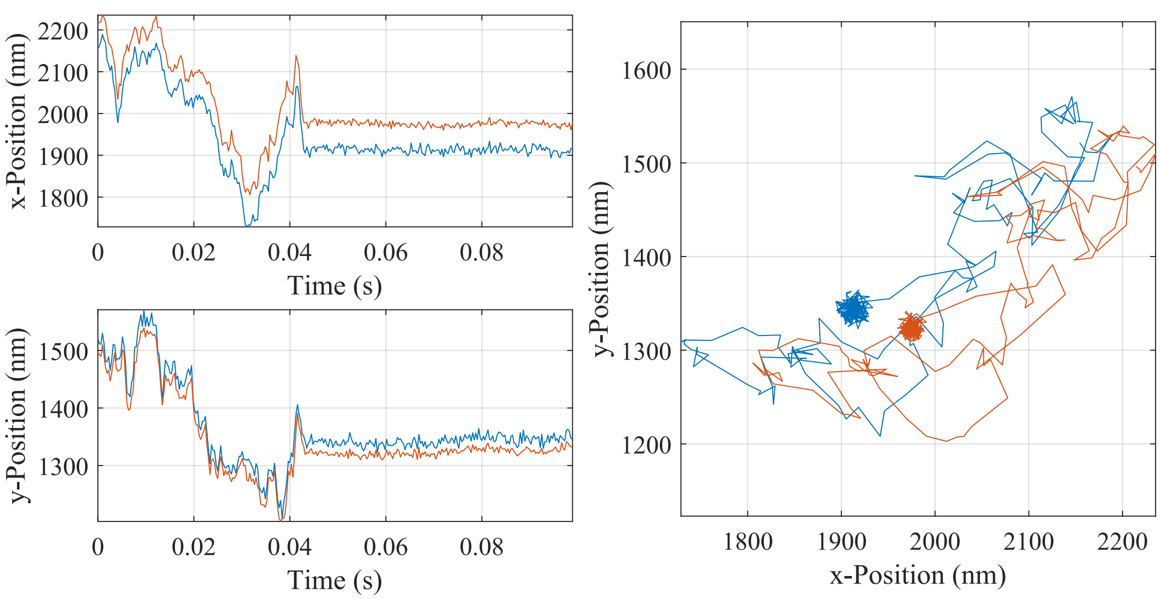

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

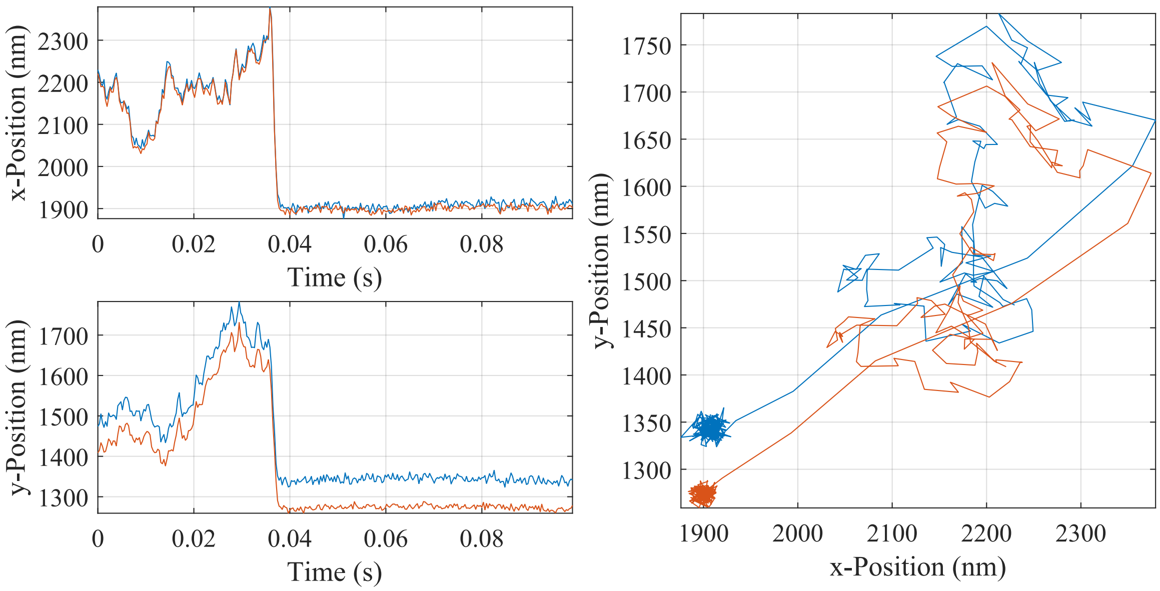

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

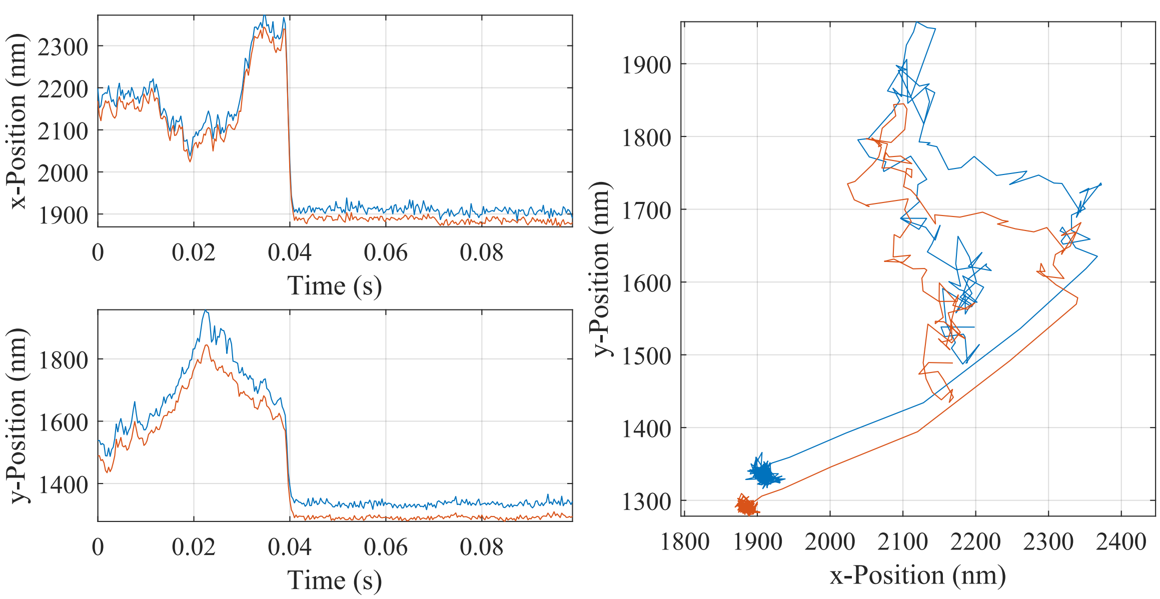

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1300
1200
1100
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
900
1900
2000
2100
2200
2300
2400
2500
x-Position (nm)

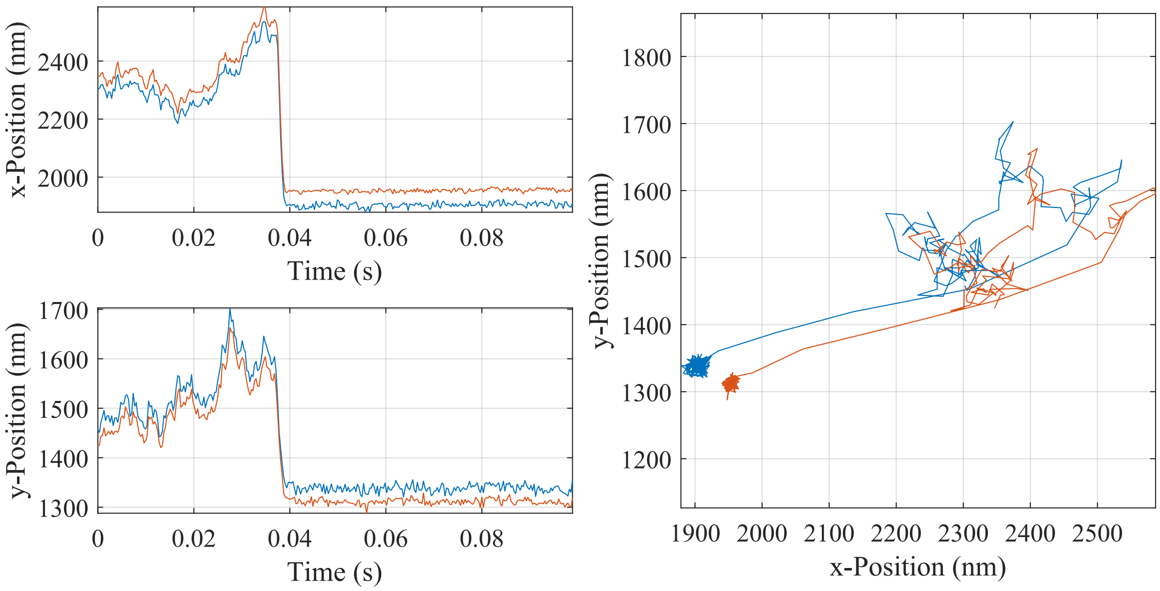

x-Position (nm)
2200
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1400
1300
1200
1100
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1450
1400
1350
1300
1250
1200
1150
1100
1900
2000
2100
2200
x-Position (nm)

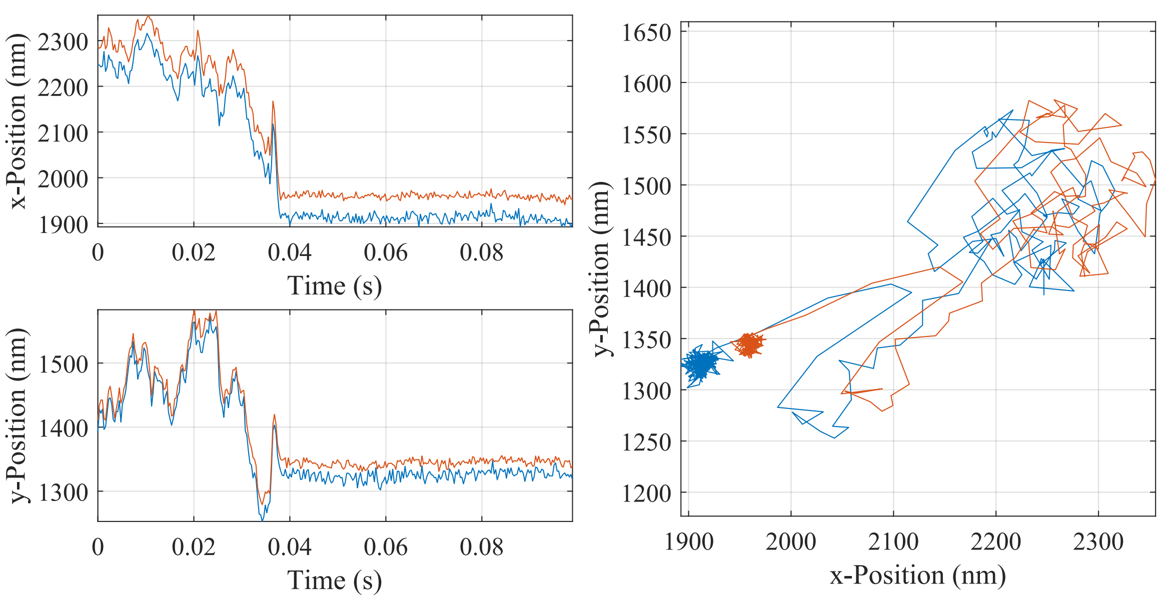

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

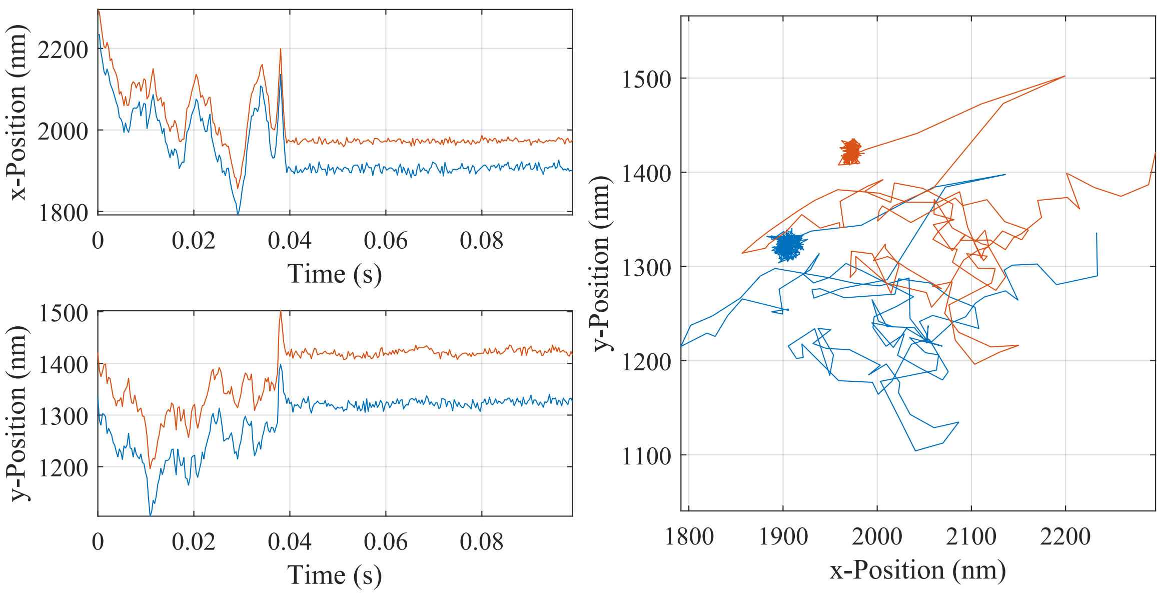

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

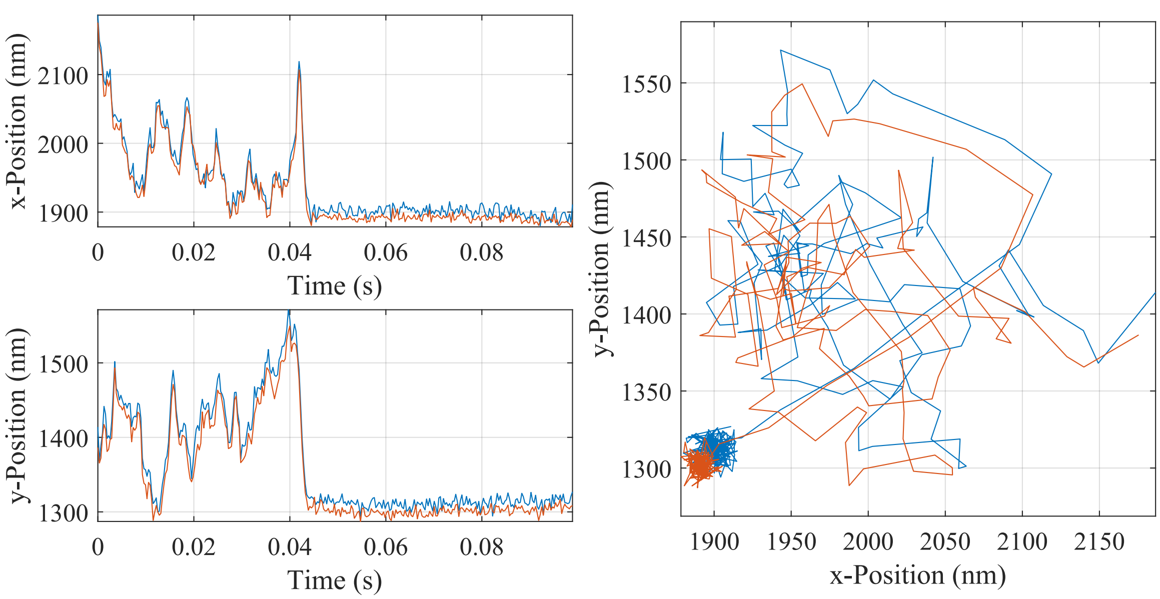

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

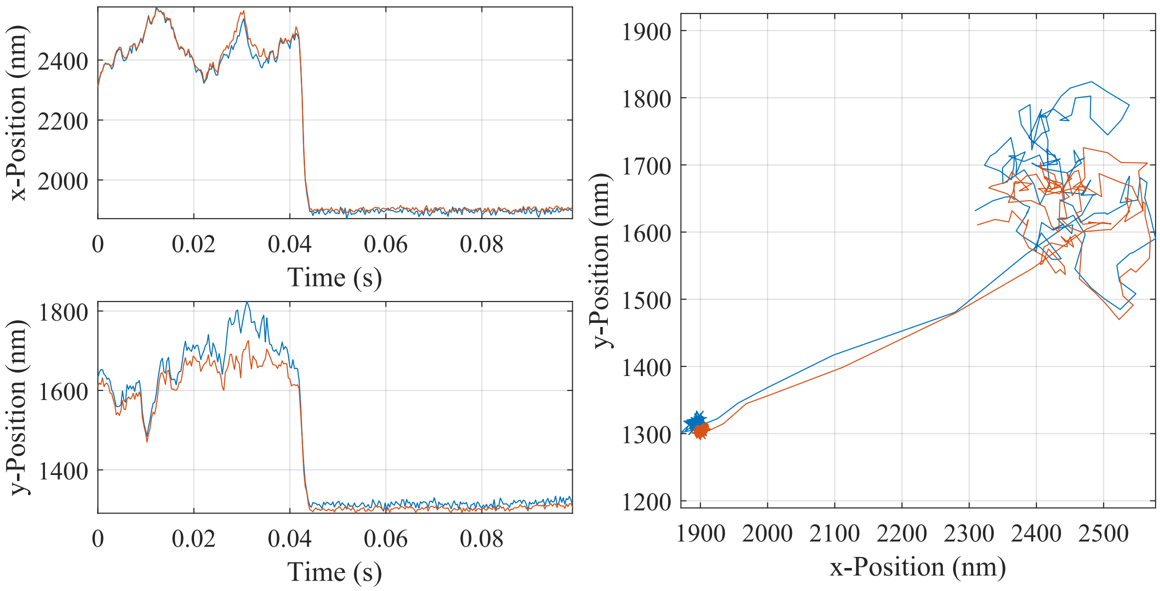

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

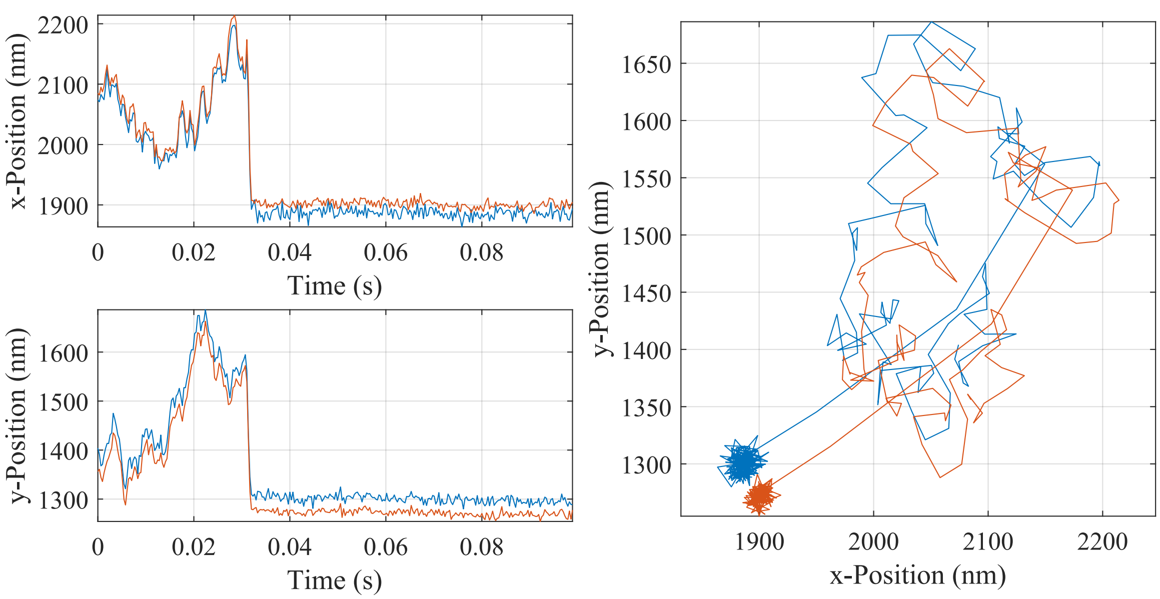

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

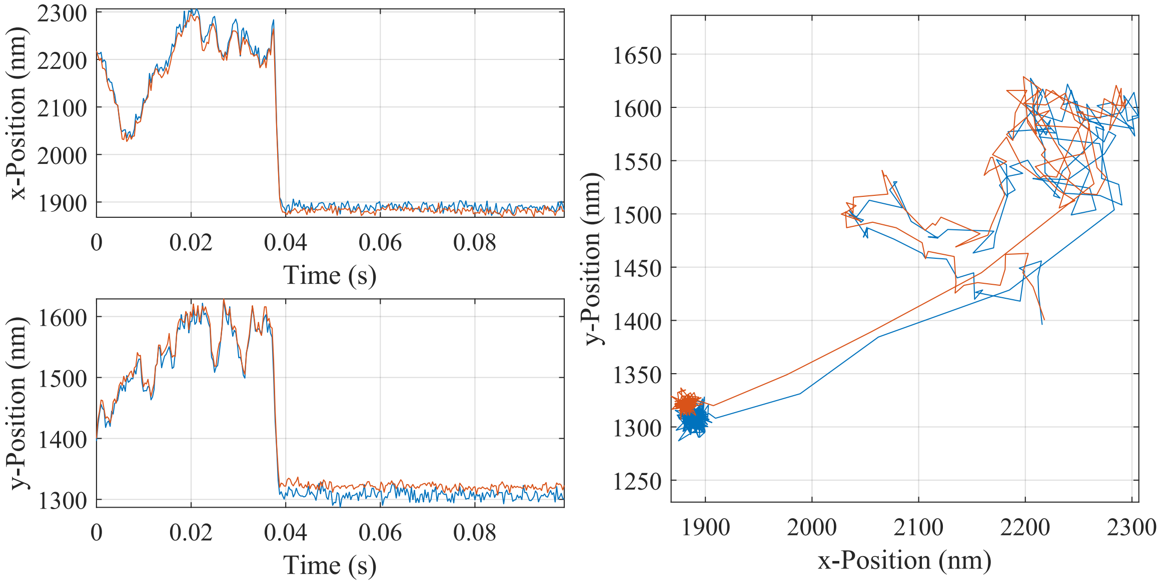

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

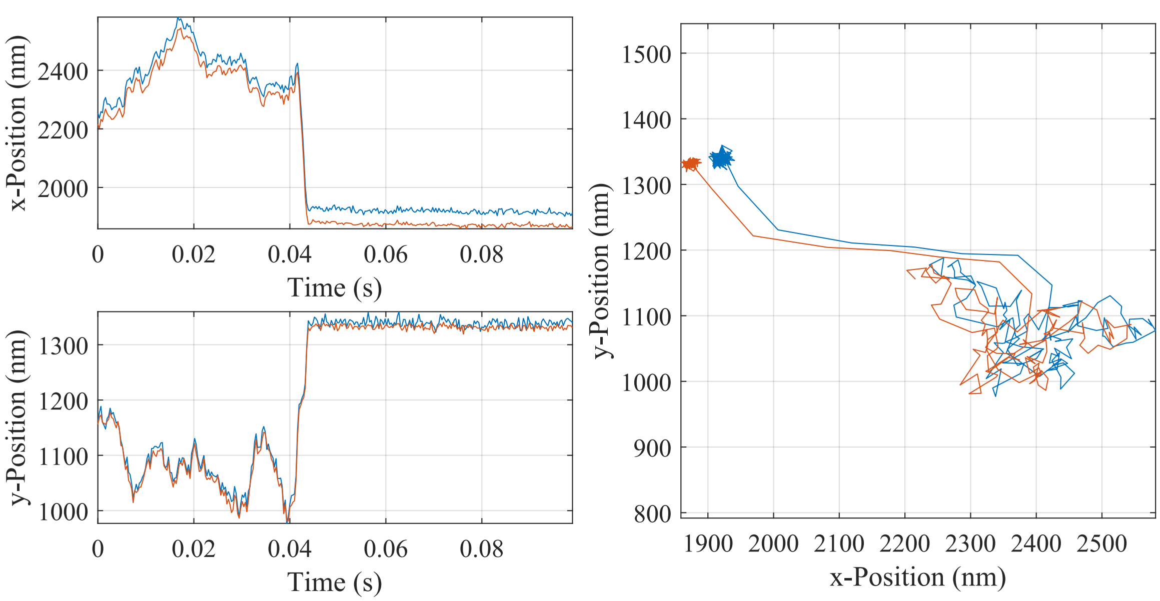

x-Position (nm)
2200
2000
1800
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
1200
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1800
1900
2000
2100
2200
x-Position (nm)

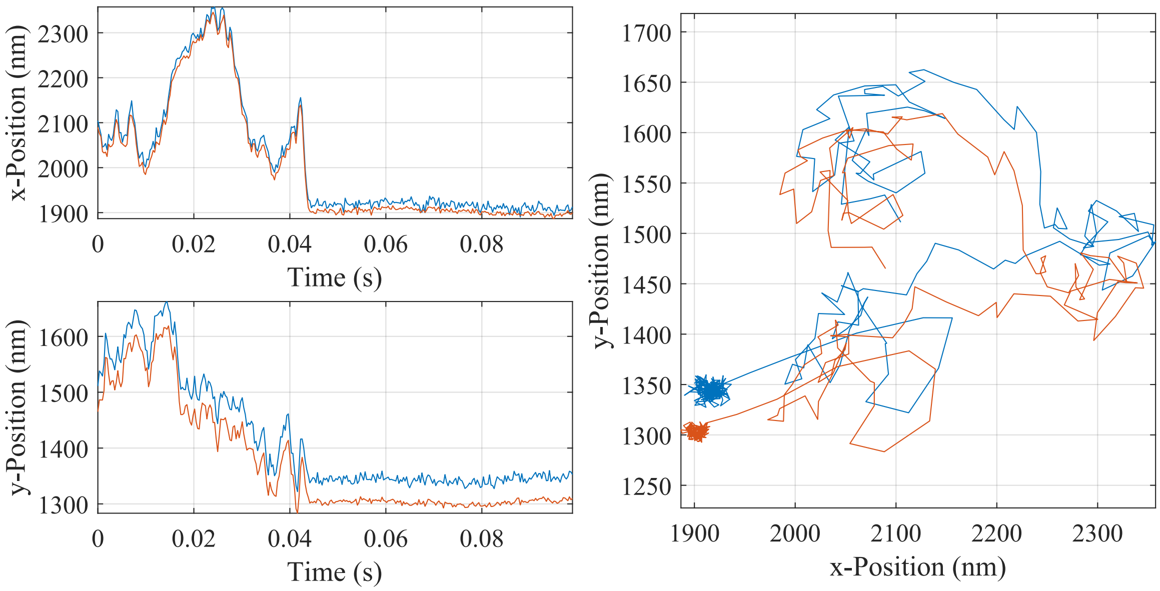

x-Position (nm)
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1550
1500
1450
1400
1350
1300
1900
1950
2000
2050
2100
2150
x-Position (nm)

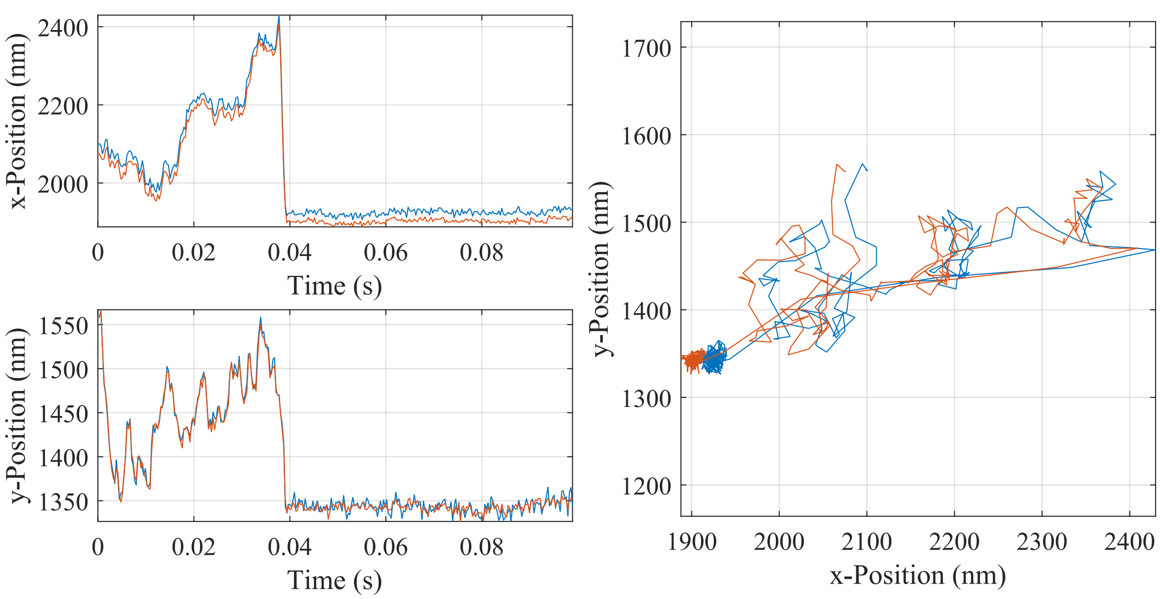

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

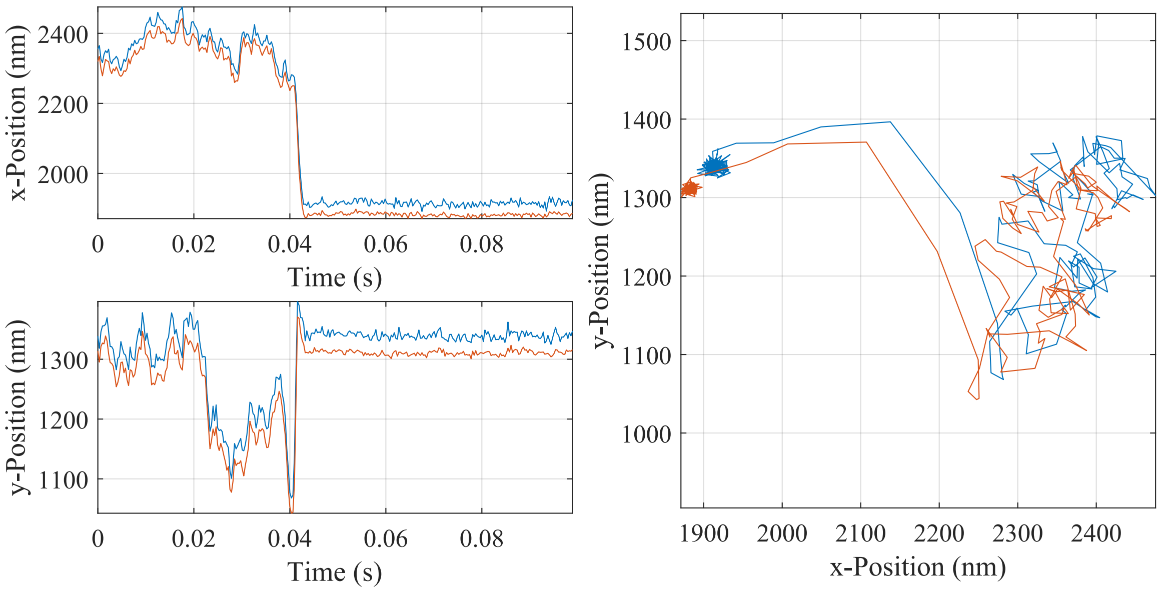

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

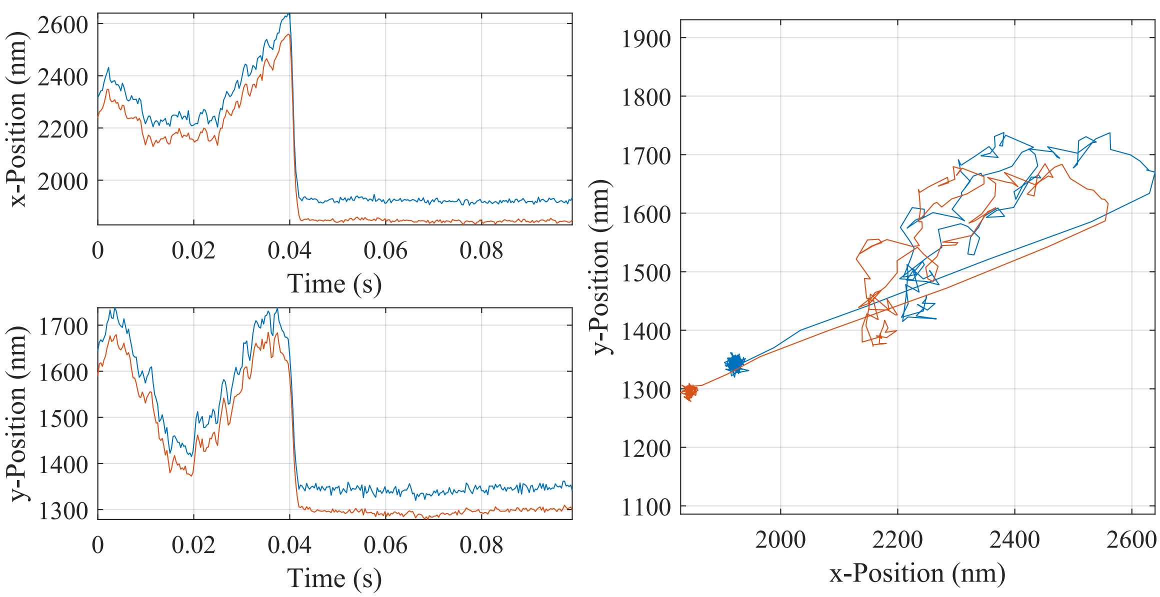

x-Position (nm)
2300
2200
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1600
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1650
1600
1550
1500
1450
1400
1350
1300
1250
1900
2000
2100
2200
2300
x-Position (nm)

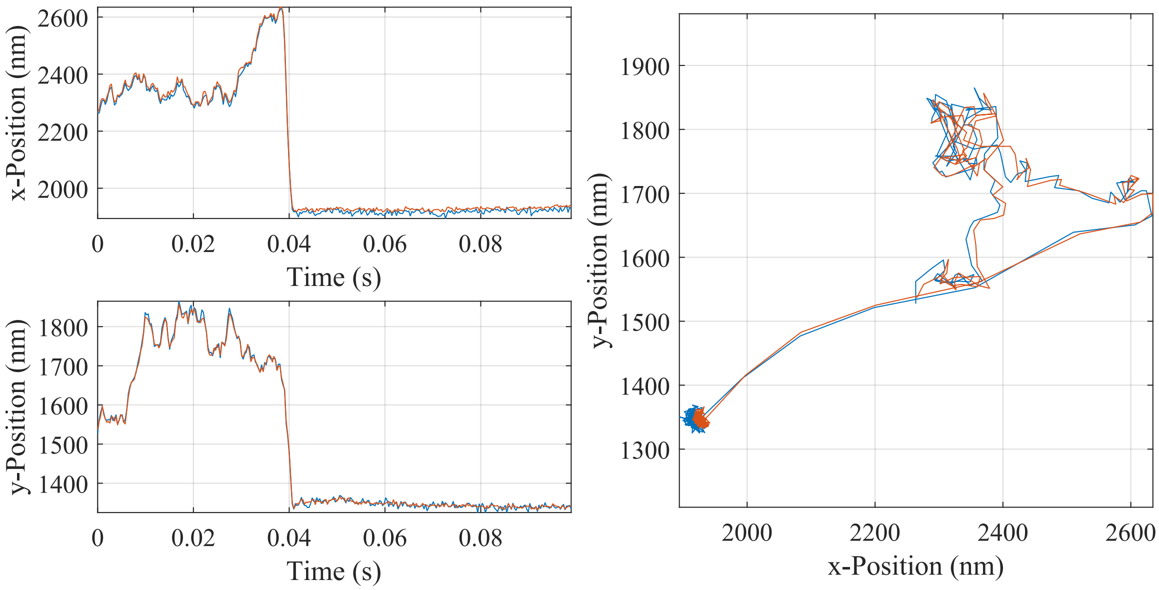

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1300
1200
1100
1000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
900
800
1900
2000
2100
2200
2300
2400
2500
x-Position (nm)

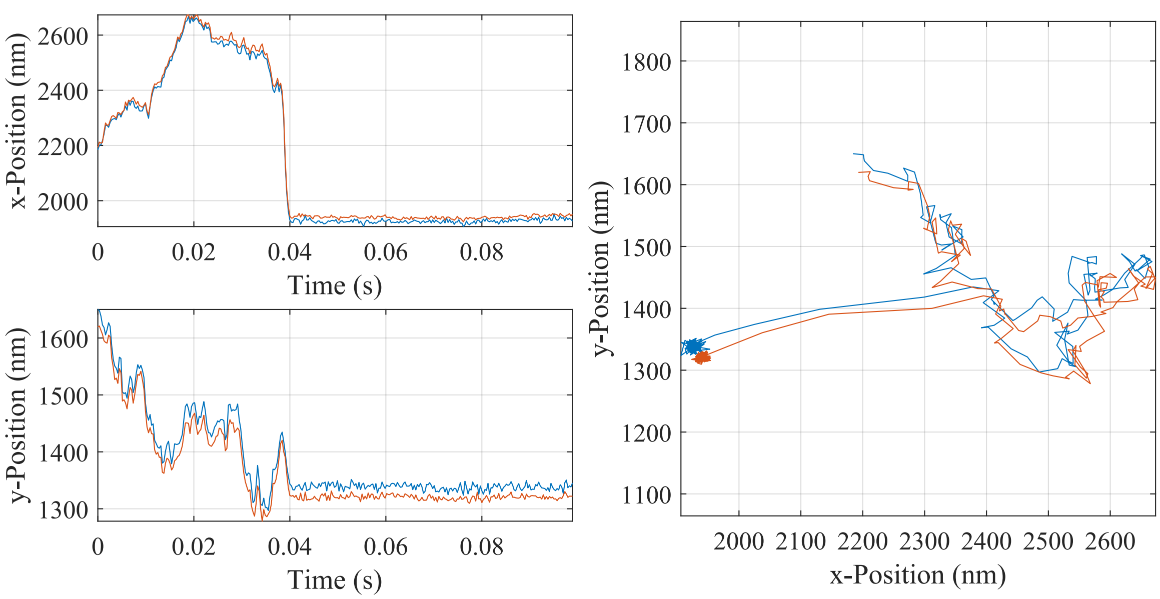

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

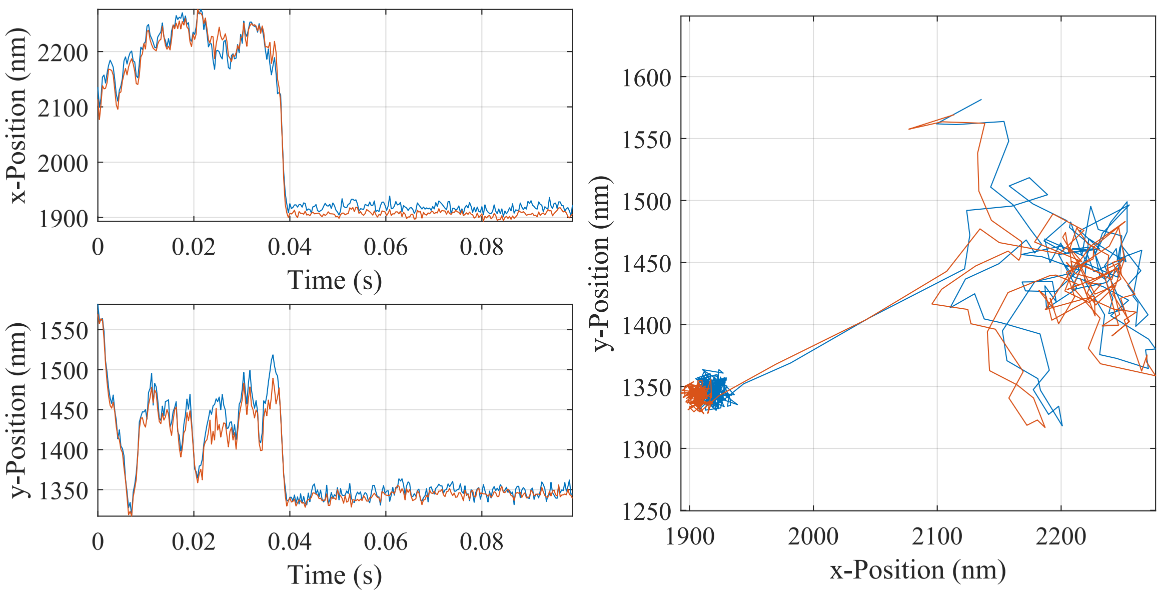

x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)

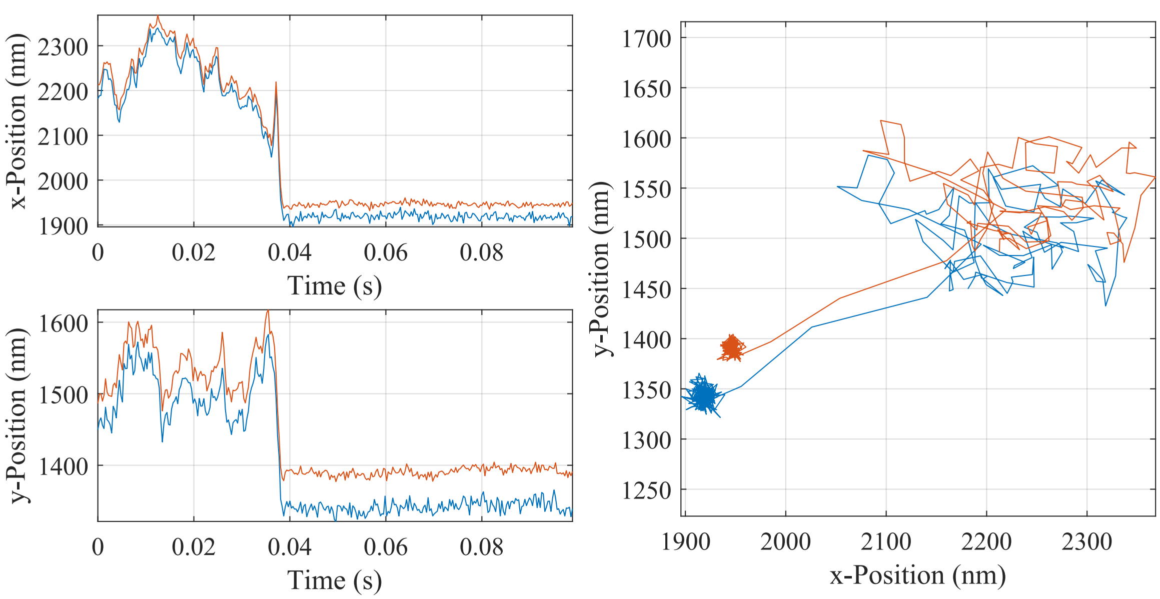

x-Position (nm)
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1300
1200
1100
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
1200
1100
1000
1900
2000
2100
2200
2300
2400
x-Position (nm)

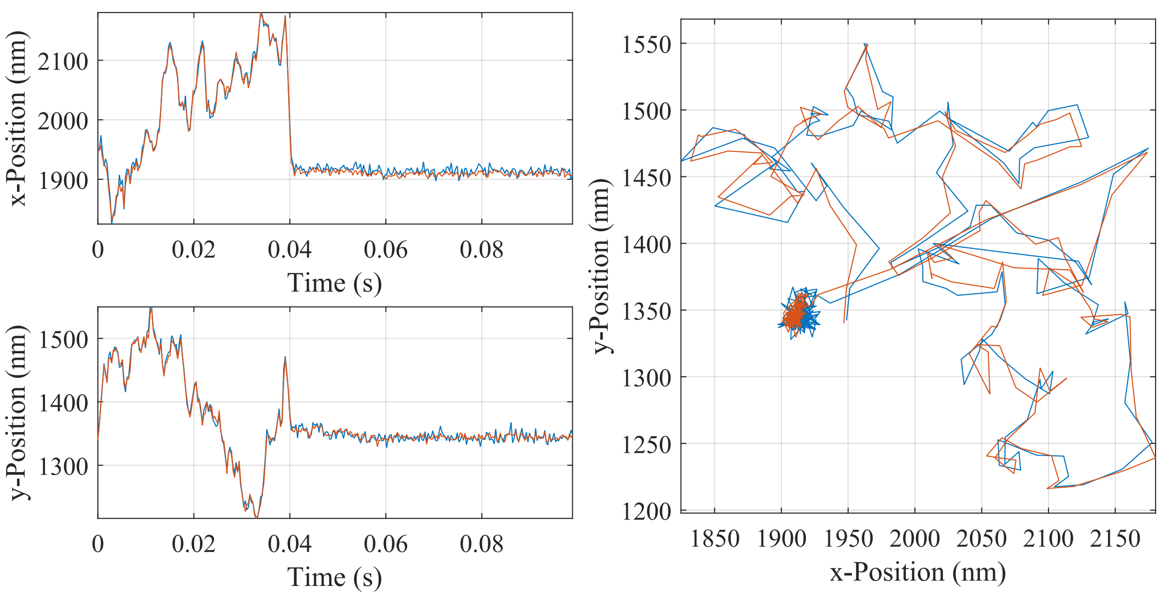

x-Position (nm)
2600
2400
2200
2000
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1700
1600
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1900
1800
1700
1600
1500
1400
1300
1200
1100
2000
2200
2400
2600
x-Position (nm)


x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)


x-Position (nm)
Time (s)
y-Position (nm)
Time (s)
y-Position (nm)
x-Position (nm)


x-Position (nm)
2200
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1550
1500
1450
1400
1350
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1600
1550
1500
1450
1400
1350
1300
1250
1900
2000
2100
2200
x-Position (nm)


x-Position (nm)
2300
2200
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1600
1500
1400
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1700
1650
1600
1550
1500
1450
1400
1350
1300
1250
1900
2000
2100
2200
2300
x-Position (nm)



x-Position (nm)
2100
2000
1900
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1500
1400
1300
0
0.02
0.04
0.06
0.08
Time (s)
y-Position (nm)
1550
1500
1450
1400
1350
1300
1250
1200
1850
1900
1950
2000
2050
2100
2150
x-Position (nm)

APPENDIX F

Code Listings for the Optical Trapping Simulation

The code presented here relies on a bunch of packages from `Julia` programming language as well as a third-party toolbox for `MATLAB`. Following is the complete list of these,

- Packages from `Julia`:
  - `DifferentialEquations.jl`: Provides a big set of Stichastic Differential Equation solvers, some of which are used here.
  - `Interpolations.jl`: Provides fast interpolation functionality, needed here to interpolate the laser force values.
  - `LinearAlgebra.jl`: Provides functions and structures for matrix operations.
  - `MATLAB.jl`: Used to communication with `MATLAB`, where the laser force calculation happens.
  - `MAT.jl`: Used to store data in a `MATLAB` readable file format, `*.mat`.
  - `Glob.jl`: Provides functionality of file name matching and pattern finding.
- Toolbox for `MATLAB`:
  - `ott`: Full name is `Optical Tweezer Toolbox`, available at `https://github.com/ilent2/ott`. It calculates the laser beam forces using T-Matrix approach of solving Mie Scattering problems. Used here to precaculate the force field created by the trapping laser.

F.1 Directory tree

/
- `expData/` .... Contains particle trajectory data
- `inputData/` .... Contains data and parameters for simulation
  - `forceFields` .... Contains files that store precomputed laser force data
  - `inputSc0.jl` .... File containing simulation parameters
- `otModule/` .... Module for optical tweezer simulation
  - `compBeamForce.jl` .... Precomputes laser beam force field using `ott`
  - `der.jl` .... Functions to calculate state derivatives
  - `opticalTweezer.jl` .... Main file creating the module
  - `sysInfo.jl` .... Defines a structure to store simulation parameters
- `results/` .... Folder to store simulation results (Automatically created)
- `beadSimCompScaling.jl` .... Effect of different scaling factors on an ensemble problem
- `beadSimCompScalingPSD.jl` .... Effect of different scaling factors on the PSD
- `beadSimCompScalingSingle.jl` .... Effect of different scaling factors on a single trajectory

F.2 Code Listings

F.2.1 `inputSc0.jl`

```julia
# Load system parameters, initial comditions and experimental data
params = sysInfo(kgram = 1e15, meter = 1e3, second = 1e3, kelvin = 1, temp = 293.15, ρₛ = 2000, rₛ = 1e-6, ρₘ = 998.2071, μₘ = 0.001002, nₛ = 1.45, nₘ = 1.33, NA = 1.2, λ₀ = 1064e-9, sf = 1);
P = 300 * 1e-3 * (params.kgram * params.meter^2 / params.second^3);
expNo = 1;

# Load experimental data
expFile = matopen("./expData/tek" * string(expNo, base = 10, pad = 4) * ".mat"); # The dataset has time in ms and displacement in mm
qpdTrTime = vec(read(expFile, "qpdTrTime")) * 1e-3 * params.second;
vidTrTime = vec(read(expFile, "vidTrTime")) * 1e-3 * params.second;
qpdTrData = read(expFile, "qpdTrData") * 1e-3 * params.meter;
vidTrData = read(expFile, "vidTrData") * 1e-3 * params.meter;
close(expFile);

# Define initial condition and time span
expInitIdx = 89;
# q123 = [vidTrpData(1,1) vidTrpData(1,2) -3.023*lambda0];
q123 = [qpdTrData[expInitIdx, 1]; qpdTrData[expInitIdx, 2]; -0 * params.λ₀];
qd123 = [0; 0; 0] * params.meter / params.second; # Initial speed
w = 0; # Work done
q0 = [q123; qd123; w]; # Initial state vector
dt = 2e-6 * params.second; # m/beta_v/5;
```

```julia
tVals = Vector(qpdTrTime[expInitIdx]:dt:qpdTrTime[end]);
relaxationTime = params.mₛ / params.βᵥ;
subInter = dt / (ceil(dt / relaxationTime * 5) - 1);
```

F.2.2 compBeamForce.jl

```julia
struct forceVals
    Z :: Matrix{<:Real}
    fVals :: Array{<:Real,3}
    tVals :: Array{<:Real,3}
    X :: Matrix{<:Real}
    fStiffness :: Vector{<:Real}
    k :: Matrix{<:Real}
    x :: Vector{<:Real}
    z :: Vector{<:Real}

    function forceVals(matPath::String)
        matStruct = matread(matPath);

        @assert haskey(matStruct,"Z") && haskey(matStruct,"fVals") &&
        ↪  haskey(matStruct,"tVals") && haskey(matStruct,"X") &&
        ↪  haskey(matStruct,"fStiffness") && haskey(matStruct,"k") &&
        ↪  haskey(matStruct,"x") && haskey(matStruct,"z")
        nx = length(matStruct["x"]); nz = length(matStruct["z"]);
        @assert size(matStruct["Z"],1) == nz && size(matStruct["X"],1) == nz &&
        ↪  size(matStruct["fVals"],1) == nz && size(matStruct["tVals"],1) == nz
        @assert size(matStruct["Z"],2) == nx && size(matStruct["X"],2) == nx &&
        ↪  size(matStruct["fVals"],2) == nx && size(matStruct["tVals"],2) == nx
        @assert length(matStruct["fStiffness"]) == 3 && size(matStruct["k"]) == (6,6)


        ↪  new(matStruct["Z"],matStruct["fVals"],matStruct["tVals"],matStruct["X"],vec(matStruct["fStiffness"
    end
end

function compBeamForce(params::sysInfo,P::Union{Vector{<:Real},Real})

    if !ispath("./inputData/forceFields/")
        mkpath("./inputData/forceFields/");
    end

    mlSessionSet = false;

    for pwr in P
        if !isfile("./inputData/forceFields/" * string(trunc(Int,params.rₛ*2e6)) * "." *
        ↪  string(trunc(Int,pwr*1e-9)) * ".mat")
            println("Computing force field for $pwr mW power.")
            if !mlSessionSet
                global s = MSession();
```

```
                put_variable(s, :kgram, mxarray(params.kgram));
                put_variable(s, :meter, mxarray(params.meter));
                put_variable(s, :second, mxarray(params.second));
                put_variable(s, :r, mxarray(params.rₛ));
                put_variable(s, :NA, mxarray(params.NA));
                put_variable(s, :n_s, mxarray(params.nₛ));
                put_variable(s, :n_m, mxarray(params.nₘ));
                put_variable(s, :lambda0, mxarray(params.λ₀));
                put_variable(s, :c, mxarray(params.c));
                mlSessionSet = true;
            end
            put_variable(s, :P, mxarray(pwr));
            mat"
            % Generate the beam in SI units
            beam = ott.BscPmGauss('NA',NA,'polarisation',[1 1i],'power',P/(kgram*meter^2/second^3),...
            'index_medium',n_m,'wavelength0',lambda0/(meter));
            % Calculate T-matrix in SI units
            shape = ott.shapes.Shape.simple('sphere',r/(meter));
            T = ott.Tmatrix.simple(shape, 'index_medium', n_m, ...
            'index_particle', n_s, 'wavelength0', lambda0/(meter));

            % Compute the forces at different locations of the beam
            z = (-4:0.01:1)*lambda0;
            x = (0:0.01:3)*lambda0;
            [X,Z] = meshgrid(x,z);
            fVals = zeros([size(Z),3]);
            tVals = zeros([size(Z),3]);
            parfor j = 1:size(Z,2)
                [force,torque] = ott.forcetorque(beam, T,'position',[X(:,j) zeros(size(Z,1),1) Z(:,j)]'/(meter));
                fVals(:,j,:) = (n_m*P/c)*reshape(force',size(Z,1),1,3);
                tVals(:,j,:) = (lambda0*P/c)*reshape(torque',size(Z,1),1,3);
            end

            % Compute trap stiffness
            [~,~,k] = ott.trap_stiffness(beam,T);
            fStiffness = abs(diag(-k(1:3,1:3)*(n_m*P/c)/meter)); % Unit - pN/mm

            % Save data to a file
            save(['./inputData/forceFields/' num2str(r*2e6) '.' num2str(P*1e-9) '.mat'],'Z','X','z','x','fVals','tVals','k','fStiffness');
            "
        end
    end
    if mlSessionSet
        eval_string(s,"delete(gcp('nocreate'))");
        close(s);
    end
end
```

F.2.3 der.jl

```julia
"""

"""

function der_f!(ẋ::Vector{<:Real}, x::Vector{<:Real}, sysData::Tuple{sysInfo,Any,Any,Any}, t::Real)
    params = sysData[1]
    fx = sysData[2]
    fy = sysData[3]
    fz = sysData[4]

    ẋ[1:3] = x[4:6]

    # Convert cartesian coordinates to cylindrical
    ϕ = atan(x[2], x[1])
    rMat = [cos(ϕ) -sin(ϕ) 0; sin(ϕ) cos(ϕ) 0; 0 0 1]
    qCyl = rMat' * x[1:3]

    # Calculate the force on the particle
    F_beam = rMat * [fx(qCyl[1], qCyl[3]); fy(qCyl[1], qCyl[3]); fz(qCyl[1], qCyl[3])]

    # Calculation of translational acceleration
    F_drag = -params.βᵥ * x[4:6]
    F_bg = (params.mₛ - params.Vₛ * params.ρₘ) * [0; 0; -params.g] # [0;0;params.rho_m*params.g*params.Vol] - params.m*[0;0;params.g]
    F = F_bg + F_beam + F_drag
    ẋ[4:6] = params.sf * params.M⁻¹ * F

    if any(isnan.(ẋ))
        throw(error("ẋ has NaN values."))
    end
end

function der_g!(ẋ::Vector{<:Real}, x::Vector{<:Real}, sysData::Tuple{sysInfo,Any,Any,Any}, t::Real)
    params = sysData[1]

    val = params.sf * params.σᵥ / params.mₛ

    ẋ[1] = 0
    ẋ[2] = 0
    ẋ[3] = 0
    ẋ[4] = val
    ẋ[5] = val
    ẋ[6] = val
end
```

F.2.4 opticalTweezer.jl

```julia
module opticalTweezer

using MATLAB
using MAT
using LinearAlgebra

export forceVals, compBeamForce, myinterp, sysInfo, der_f!, der_g!

include("myinterp.jl")
include("sysInfo.jl")
include("compBeamForce.jl")
include("der.jl")

end
```

F.2.5 sysInfo.jl

```julia
struct sysInfo
    # All parameters values are defined with SI(kgram, meter, second) unit system.
    kgram :: Real; meter :: Real; second :: Real; kelvin :: Real;
    temp :: Real # System Temperature
    k□ :: Real# Boltzmann Constant
    ρₛ :: Real # Bead density
    rₛ :: Real # Bead radius 0.5e-3*(meter);#
    Vₛ :: Real # Bead volume
    mₛ :: Real # Bead mass
    M⁻¹ :: Matrix{<:Real} # Inverse of mass matrix
    I :: Matrix{<:Real} # Inertia matrix
    g :: Real # Gravitational acceleration
    ρₘ :: Real # Density of fluid medium
    μₘ :: Real # Dynamic viscosity of fluid medium at 20d C
    βᵥ :: Real # Translational drag coefficient
    βω :: Real # Rotational drag coefficient
    σᵥ :: Real # Standard deviation for translational Brownian motion
    σω :: Real # Standard deviation for rotational Brownian motion
    nₛ :: Real # Particle's refractive index
    nₘ :: Real # Medium's refractive index
    c :: Real # Speed of light in a vaccum
    NA :: Real # Numerical Aperture of objective
    λ₀ :: Real # Wavelength of laser
    sf :: Real # Scaling factor

    """

        sysInfo(;kgram,meter,second,kelvin,temp,ρₛ,rₛ,ρₘ,μₘ,nₛ,nₘ,NA,λ₀,sf)

```

```julia
Creates a `sysInfo` structure to store all the necessary Optical Tweezer simulation parameters.

# Example
```julia> sysInfo(kgram = 1e15, meter = 1e3, second = 1e3, kelvin = 1, temp = 293.15, ρₛ = 1050, rₛ = 0.25e-6, ρₘ = 998.2071, μₘ = 0.001002, nₛ = 1.57773, nₘ = 1.33, NA = 1.3, λ₀ = 8e-7, sf = 1)```

All parameters values should be provided in SI(kgram, meter, second) unit system.

Conversion to desired unit system will be handeled using `kgram`, `meter`, `second` and `kelvin` parameters at the time of object construction.

# Member/Parameter explanations
- `kgram` : Unit conversion constant for kg. If the unit system uses mg then this value would be 1e6.
- `meter` : Unit conversion constant for m. If the unit system uses nm then this value would be 1e9.
- `second` : Unit conversion constant for s. If the unit system uses μs then this value would be 1e6.
- `kelvin` : Unit conversion constant for °K. This is generally 1.
- `temp` : System Temperature in °K
- `kᵦ` : Boltzmann Constant
- `ρₛ` : Bead density
- `rₛ` : Bead radius
- `Vₛ` : Bead volume = `4/3*π*rₛ*rₛ*rₛ`
- `mₛ` : Bead mass = `ρₛ*Vₛ`
- `M⁻¹` : Inverse of mass matrix = `diagm([1/mₛ, 1/mₛ, 1/mₛ])`
- `I` : Inertia matrix = `diagm([2/5*mₛ*rₛ*rₛ, 2/5*mₛ*rₛ*rₛ, 2/5*mₛ*rₛ*rₛ])`
- `g` : Gravitational acceleration = `9.80665*(meter/second^2)`
- `ρₘ` : Density of fluid medium
- `μₘ` : Dynamic viscosity of fluid medium at defined `temp`
- `βᵥ` : Translational drag coefficient = `6*π*μₘ*rₛ`
- `βω` : Rotational drag coefficient = `8*π*μₘ*rₛ^3`
- `σᵥ` : Standard deviation for translational Brownian motion = `sqrt(2*βᵥ*kᵦ*temp)`
- `σω` : Standard deviation for rotational Brownian motion = `sqrt(2*βω*kᵦ*temp)`
- `nₛ` : Particle's refractive index
- `nₘ` : Medium's refractive index
- `c` : Speed of light in a vaccum
- `NA` : Numerical Aperture of objective
- `λ₀` : Wavelength of laser
- `sf` : Scaling factor
"""
function sysInfo(;kgram::Real,meter::Real,second::Real,kelvin::Real,temp::Real,ρₛ::Real,rₛ::Real,ρₘ::Real,μₘ::Re
    temp = temp*kelvin;
    kᵦ = 1.38064852*1e-23*(meter^2*kgram/second^2/kelvin);
    ρₛ = ρₛ*(kgram/meter^3);
    rₛ = rₛ*(meter);
    Vₛ = 4/3*π*rₛ*rₛ*rₛ;
    mₛ = ρₛ*Vₛ;
```
```

```
        M⁻¹ = diagm([1/mₛ, 1/mₛ, 1/mₛ]);
        I = diagm([2/5*mₛ*rₛ*rₛ, 2/5*mₛ*rₛ*rₛ, 2/5*mₛ*rₛ*rₛ]);
        g = 9.80665*(meter/second^2);
        ρₘ = ρₘ*(kgram/meter^3);
        μₘ = μₘ*(kgram/meter/second);
        βᵥ = 6*π*μₘ*rₛ;
        βω = 8*π*μₘ*rₛ^3;
        σᵥ = sqrt(2*βᵥ*k□*temp);
        σω = sqrt(2*βω*k□*temp);

        new(
            kgram,meter,second,kelvin,
            temp*kelvin,
            k□,
            ρₛ,
            rₛ,
            Vₛ,
            mₛ,
            M⁻¹,
            I,
            g,
            ρₘ,
            μₘ,
            βᵥ,
            βω,
            σᵥ,
            σω,
            nₛ,
            nₘ,
            299792458*(meter/second),
            NA,
            λ₀*(meter),
            sf
        )
    end
end

function Base.print(io::Core.IO, params::sysInfo)
    for fname in fieldnames(typeof(params))
        println("$fname = $(getfield(params, fname))")
    end
end

Base.print(params::sysInfo) = print(stdout,params)

Base.println(io::Core.IO,params::sysInfo) = println(io,params)

Base.println(params::sysInfo) = println(stdout,params)
```

F.2.6 beadSimCompScaling.jl

```julia
include("./otModule/opticalTweezer.jl")
using .opticalTweezer
using DifferentialEquations
using Interpolations
using Glob
using MAT
using DifferentialEquations.EnsembleAnalysis

for i in 0:3
    include("./inputData/inputSc" * string(i) * ".jl")

    # Compute the force field and load the data
    compBeamForce(params, P)
    forceData = forceVals("./inputData/forceFields/" * string(trunc(Int, params.rₛ * 2e6)) * "." * string(trunc(Int, P * 1e-9)) * ".mat")
    fx = scale(interpolate(forceData.fVals[:, :, 1]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
    fy = scale(interpolate(forceData.fVals[:, :, 2]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
    fz = scale(interpolate(forceData.fVals[:, :, 3]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)

    # Run the SDE solver
    prob = SDEProblem(der_f!, der_g!, q0[1:(end-1)], (tVals[1], 30), (params, fx, fy, fz))
    ensembleprob = EnsembleProblem(prob)
    compTime = @elapsed begin
        sol = solve(ensembleprob, ImplicitEM(), EnsembleThreads(), trajectories=10000)
    end
    summ = EnsembleSummary(sol)
    compTime = @elapsed begin
        sol = solve(ensembleprob, ImplicitEM(), EnsembleThreads(), trajectories=10000, saveat=summ.t)
    end
    qMean, qVar = timeseries_steps_meanvar(sol)
    qMean = Array(qMean[:, :]')
    qVar = Array(qVar[:, :]')
    t = Array(sol.u[1].t)
    println(compTime)

    # Save the data to a mat file
    if !ispath("./results/ensemble/")
        mkpath("./results/ensemble/")
    end
    rm.(glob("./results/ensemble/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat"))
    matwrite("./results/ensemble/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat", Dict(
        "expNo" => expNo,
        "kgram" => params.kgram,
```

```
        "meter" => params.meter,
        "second" => params.second,
        "kelvin" => params.kelvin,
        "temp" => params.temp,
        "rho_s" => params.ρₛ,
        "r_s" => params.rₛ,
        "rho_m" => params.ρₘ,
        "mu_m" => params.μₘ,
        "n_s" => params.nₛ,
        "n_m" => params.nₘ,
        "NA" => params.NA,
        "P" => P,
        "lambda_0" => params.λ₀,
        "sf" => params.sf,
        "expInitIdx" => expInitIdx,
        "t" => t,
        "qMean" => qMean,
        "qVar" => qVar,
        "compTime" => compTime
    ))

end
```

F.2.7 beadSimCompScalingPSD.jl

```
include("./otModule/opticalTweezer.jl")
using .opticalTweezer
using DifferentialEquations
using Interpolations
using Glob
using MAT

for i in 0:3
    include("./inputData/inputSc" * string(i) * ".jl")

    # Compute the force field and load the data
    compBeamForce(params, P)
    forceData = forceVals("./inputData/forceFields/" * string(trunc(Int, params.rₛ * 2e6))
    ↪  * "." * string(trunc(Int, P * 1e-9)) * ".mat")
    fx = scale(interpolate(forceData.fVals[:, :, 1]', BSpline(Quadratic(Line(OnGrid())))),
    ↪  (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
    fy = scale(interpolate(forceData.fVals[:, :, 2]', BSpline(Quadratic(Line(OnGrid())))),
    ↪  (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
    fz = scale(interpolate(forceData.fVals[:, :, 3]', BSpline(Quadratic(Line(OnGrid())))),
    ↪  (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)

    # Run the SDE solver
    prob = SDEProblem(der_f!, der_g!, q0[1:(end-1)], (0, 11000), (params, fx, fy, fz))
    compTime = @elapsed begin
```

```
        sol = solve(prob, ImplicitEM(), dtmax=0.001, save_idxs=[1, 2], saveat=0.001, maxiters=1e8)
    end
    t = sol.t[1001:end]
    q = Array(sol[:, 1001:end]')
    println(compTime)

    # Save the data to a mat file
    if !ispath("./results/PSD/")
        mkpath("./results/PSD/")
    end
    rm.(glob("./results/PSD/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat"))
    matwrite("./results/PSD/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat", Dict(
        "expNo" => expNo,
        "kgram" => params.kgram,
        "meter" => params.meter,
        "second" => params.second,
        "kelvin" => params.kelvin,
        # "temp" => params.temp,
        # "rho_s" => params.ρₛ,
        # "r_s" => params.rₛ,
        # "rho_m" => params.ρₘ,
        # "mu_m" => params.μₘ,
        # "n_s" => params.nₛ,
        # "n_m" => params.nₘ,
        # "NA" => params.NA,
        # "P" => P,
        # "lambda_0" => params.λ₀,
        # "sf" => params.sf,
        # "expInitIdx" => expInitIdx,
        "t" => t,
        "q" => q,
        "compTime" => compTime
    ))
end
```

F.2.8 beadSimCompScalingSingle.jl

```
include("./otModule/opticalTweezer.jl")
using .opticalTweezer
using DifferentialEquations
using Interpolations
using Plots
using Glob
using MAT

for i in 0:3
```

```
include("./inputData/inputSc" * string(i) * ".jl")

# Compute the force field and load the data
compBeamForce(params, P)
forceData = forceVals("./inputData/forceFields/" * string(trunc(Int, params.rₛ * 2e6)) * "." * string(trunc(Int, P * 1e-9)) * ".mat")
fx = scale(interpolate(forceData.fVals[:, :, 1]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
fy = scale(interpolate(forceData.fVals[:, :, 2]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)
fz = scale(interpolate(forceData.fVals[:, :, 3]', BSpline(Quadratic(Line(OnGrid())))), (0:0.01:4) * params.λ₀, (-4:0.01:4) * params.λ₀)

# Run the SDE solver
prob = SDEProblem(der_f!, der_g!, q0[1:(end-1)], (tVals[1], tVals[end]), (params, fx, fy, fz))
compTime = @elapsed begin
    sol = solve(prob, ImplicitEM(), dtmax=0.015)
end
t = sol.t
q = Array(sol[:, :]')
println(compTime)

# Save the data to a mat file
if !ispath("./results/single/")
    mkpath("./results/single/")
end
rm.(glob("./results/single/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat"))
matwrite("./results/single/" * "tek" * string(expNo, base=10, pad=4) * "." * string(params.sf) * ".mat", Dict(
    "expNo" => expNo,
    "kgram" => params.kgram,
    "meter" => params.meter,
    "second" => params.second,
    "kelvin" => params.kelvin,
    "temp" => params.temp,
    "rho_s" => params.ρₛ,
    "r_s" => params.rₛ,
    "rho_m" => params.ρₘ,
    "mu_m" => params.μₘ,
    "n_s" => params.nₛ,
    "n_m" => params.nₘ,
    "NA" => params.NA,
    "P" => P,
    "lambda_0" => params.λ₀,
    "sf" => params.sf,
    "expInitIdx" => expInitIdx,
    "t" => t,
    "q" => q,
    "compTime" => compTime
```

```
        ))
end
```

APPENDIX G

Code Listings for the Microcontroller

G.1 Directory tree

/
- CMakeLists.txt .... CMake configuration file
- helpDocs.cpp .... Strings to print as help document
- helpDocs.h .... Header file for helpDocs.cpp
- serialCOM.cpp .... Manages USB Serial communication with the PC
- serialCOM.h .... Header file for serialCOM.cpp
- pico_sdk_import.cmake ...... Find and configure Pico C/C++ SDK for CMake
- serialInput.cpp .... Parses strings from the USB Serial communication
- serialInput.h .... Header file for serialInput.cpp
- shutter.cpp .... Main code of the microcontroller

G.2 Code Listings

G.2.1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.13)

set(ENV{PICO_SDK_PATH} "~/pico/pico-sdk/")
include(pico_sdk_import.cmake)

project(hddVcShutterCode C CXX ASM)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
pico_sdk_init()

add_executable(shutter
  serialInput.cpp
  serialCOM.cpp
  helpDocs.cpp
  shutter.cpp
)

target_include_directories(shutter PRIVATE
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}
)

pico_enable_stdio_usb(shutter 1)
pico_enable_stdio_uart(shutter 0)

# Turn off floating point support
target_compile_definitions(shutter PRIVATE
PICO_DEFAULT_FLOAT_IMPL=pico_float_none
PICO_DEFAULT_DOUBLE_IMPL=pico_double_none
)

# Link common dependencies
```

```
target_link_libraries(shutter pico_stdlib hardware_timer hardware_watchdog hardware_pwm)

# Create map/bin/hex files
pico_add_extra_outputs(shutter)
```

G.2.2 helpDocs.cpp

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#include "helpDocs.h"

void help(const char *command)
{
    if (!strcmp(command, NULL))
        help_about();
    else if (!strcmp(command, "shutterForce"))
        help_shutterForce();
    else if (!strcmp(command, "openingTiming"))
        help_openingTiming();
    else if (!strcmp(command, "closingTiming"))
        help_closingTiming();
    else if (!strcmp(command, "cycleTiming"))
        help_cycleTiming();
}

void help_about()
{
    const char *doc = "\r\n\r\n"
                      "===============================================================>\r\n"
                      "Optical Tweezer Laser Shutter and Camera Trigger Controller\r\n"
                      "Author: Vatsal Asitkumar Joshi\r\n"
                      "===============================================================>\r\n"
                      "\r\n"
                      "This program is developed to control the shutter made out of a\r\n"
                      "Hard Disc Drive(HDD) and sync it with a camera to record the\r\n"
                      "motion of a bead in an optical trap.\r\n"
                      "\r\n"
                      "The user has to setup timings of one \"Experiment\" (Exp). The\r\n"
                      "shutter can be in one of the following states during an Exp:\r\n"
                      "\tExp start        ->  This is the default resting state of the\r\n"
                      "\t |                   shutter. In this state, the shutter is\r\n"
                      "\t |                   assumed to be open and the output for\r\n"
                      "\t |                   camera trigger is off. From this point\r\n"
```

```
"\t |                       onward, the user is not allowed to change\r\n"
"\t |                       any parameters until the experiment\r\n"
"\t |                       is completed.\r\n"
"\tClosing          ->  The shutter is moving towards the closed\r\n"
"\t |                       state. This state is sub-divided into\r\n"
"\t |                       three more states which are start, delay\r\n"
"\t |                       and end. In start state, a negative torque\r\n"
"\t |                       is applied on the shutter to start moving\r\n"
"\t |                       it in the closed state. Then the power to\r\n"
"\t |                       the shutter is cut-off momentarily. And\r\n"
"\t |                       finally, a positive torque is applied on\r\n"
"\t |                       the shutter to stop its motion. The sum\r\n"
"\t |                       of all 3 sub-state time periods is called\r\n"
"\t |                       \"timeShutterClosing\".\r\n"
"\tClosed           ->  Right after the \"Closing stop\" state of\r\n"
"\t |                       the shutter, a very small negative torque\r\n"
"\t |                       is applied on the shutter to keep it in the\r\n"
"\t |                       closed sate. The time period of this state\r\n"
"\t |                       is called \"timeShutterStayClosed\". Also,\r\n"
"\t |                       the camera trigger will be activated during\r\n"
"\t |                       this state.\r\n"
"\tOpening          ->  This state is very similar to the closing\r\n"
"\t |                       state with the only difference that the\r\n"
"\t |                       torque directions in each sub-state (start,\r\n"
"\t |                       delay and end) is reversed so that the\r\n"
"\t |                       shutter now moves towards the opened state.\r\n"
"\t |                       The sum of all 3 sub-state time periods is\r\n"
"\t |                       called \"timeShutterOpening\".\r\n"
"\tOpened           ->  In this state, the power to the shutter is\r\n"
"\t |                       cut-off and the system waits for the\r\n"
"\t |                       remaining time of one Exp. The time period\r\n"
"\t |                       of one complete experiment is called\r\n"
"\t |                       \"timeOfExperiment\".\r\n"
```

```c
                      "\tExp end          ->  At this point, the system moves to \"Exp start\"\r\n"
                      "\t                     state and stays there until another Exp is\r\n"
                      "\t                     initiated by the software.\r\n"
                      "\r\n"
                      "Following is the list of commands and their basic use,\r\n"
                      "\thelp             ->  To show this help document and the detailed\r\n"
                      "\t                     documentation for a specific command.\r\n"
                      "\tshutterForce     ->  Set the shutter torque amplitude as a %%\r\n"
                      "\t                     of maximum possible value for \"Closing\" or\r\n"
                      "\t                     \"Opening\" and \"Closed\" states.\r\n"
                      "\topeningTiming    ->  Set timings of the sub-states of \"Opening\"\r\n"
                      "\t                     state.\r\n"
                      "\tclosingTiming    ->  Set timings of the sub-states of \"Closing\"\r\n"
                      "\t                     state.\r\n"
                      "\tcycleTiming      ->  Set the timings of one experiment.\r\n";

    // Print out the documentation
    putsUSB(doc);
}

void help_shutterForce()
{
    const char *doc = "\tshutterForce <movingForce> <hldingForce>\r\n"
                      "\r\n"
                      "This command allows the user to set %% of the max\r\n"
                      "possible force to be used during shutter motion\r\n"
                      "states which are \"Closing\" and \"Opening\" and\r\n"
                      "the shutter holding state which is \"Closed\".\r\n"
                      "These values are set with command arguments <movingForce>\r\n"
                      "and <hldingForce> respectively. Both of these aregruments\r\n"
                      "must have values between 0 and 100 inclusive. And, these\r\n"
                      "values have to be integer. If the command is called\r\n"
                      "without any arguments then the current set values are\r\n"
                      "printed out.\r\n"
                      "\r\n"
                      "Example:\r\n"
                      "\t>shutterForce               This will print out current values of <movingForce> and <hldingForce>.\r\n"
                      "\t>shutterForce 100 10        This will make Closing/Opening torque = 100%% of max possible torque\r\n"
                      "\t                            and Closed torque = 10%% of max possible torque.\r\n"
                      "\r\n";

    // Print out the documentation
```

```
    putsUSB(doc);
}

void help_openingTiming()
{
    const char *doc = "\topeningTiming <start> <delay> <stop>\r\n"
                      "\r\n"
                      "User can set the time period of each sub-state\r\n"
                      "of \"Opening\" state using this command. The\r\n"
                      "<movingForce>%% of max possible power will be\r\n"
                      "applied for <start> microseconds to make the\r\n"
                      "shutter start moving towards \"Opened\" state.\r\n"
                      "In \"Opening\" state, the power will be cut-off\r\n"
                      "for <delay> microseconds. And, the negative\r\n"
                      "<movingForce>%% of max possible power will be\r\n"
                      "applied for <stop> microseconds to make the\r\n"
                      "shutter stop moving. All the argument values\r\n"
                      "have to be integers and are assumed to be in\r\n"
                      "the units of microseconds. If the command is\r\n"
                      "entered without any arguments then current set\r\n"
                      "values are printed out.\r\n"
                      "\r\n"
                      "Example:\r\n"
                      "\t>openingTiming                   This will print out current values of <start>, <delay> and <stop>.\r\n"
                      "\t>openingTiming 3000 2500 2000    This will make openingStart = 3ms, openingDelay = 2.5ms and openingStop = 2ms.\r\n"
                      "\r\n";

    // Print out the documentation
    putsUSB(doc);
}

void help_closingTiming()
{
    const char *doc = "\tclosingTiming <start> <delay> <stop>\r\n"
                      "\r\n"
                      "User can set the time period of each sub-state\r\n"
                      "of \"Closing\" state using this command. The\r\n"
                      "negative <movingForce>%% of max possible power\r\n"
                      "will be applied for <start> microseconds to\r\n"
                      "make the shutter start moving towards \"Closed\"\r\n"
                      "state. In \"Closing\" state, the power will be\r\n"
                      "cut-off for <delay> microseconds. And, the\r\n"
                      "<movingForce>%% of max possible power will be\r\n"
                      "applied for <stop> microseconds to make the\r\n"
                      "shutter stop moving. All the argument values\r\n"
                      "have to be integers and are assumed to be in\r\n"
                      "the units of microseconds. If the command is\r\n"
                      "entered without any arguments then current set\r\n"
                      "values are printed out."
```

```
                      "\r\n"
                      "Example:\r\n"
                      "\t>closingTiming                   This will print out current values of <start>, <delay> and <stop>.\r\n"
                      "\t>closingTiming 4000 800 2900     This will make closingStart = 4ms, closingDelay = 0.8ms and closingStop = 2.9ms.\r\n"
                      "\r\n";

    // Print out the documentation
    putsUSB(doc);
}

void help_cycleTiming()
{
    const char *doc = "\tcycleTiming <timeShutterStayClosed> <timeCameraTrigAdv> <timeCameraRecorords> <timeOfExperiment>\r\n"
                      "\r\n"
                      "This command allows to set all the necessary\r\n"
                      "parameters for one experiment. All the values\r\n"
                      "provided here are assumed to be in microseconds\r\n"
                      "and must be integer values. If this command is\r\n"
                      "entered without any arguments then current set\r\n"
                      "values will be printed out. The time of one\r\n"
                      "experiment <timeOfExperiment> is defined as\r\n"
                      "\r\n"
                      "   timeOfExperiment = max(\r\n"
                      "                          timeShutterClosing + timeShutterStayClosed + timeShutterOpening\r\n"
                      "                          ,\r\n"
                      "                          timeShutterClosing + timeShutterStayClosed + timeShutterOpening + timeCameraRecorords - timeCameraTrigAdv\r\n"
                      "                          )\r\n"
                      "\r\n"
                      "The meaning of each variable is explained below.\r\n"
                      "\ttimeShutterClosing           Time period for which shutter is in \"Closing\" state,\r\n"
                      "\t                             i.e. sum of all sub-states of the \"Closing\" state.\r\n"
                      "\ttimeShutterStayClosed        Time period for which the shutter stays in \"Closed\" state.\r\n"
                      "\ttimeShutterOpening           Time period for which shutter is in \"Opening\" state,\r\n"
                      "\t                             i.e. sum of all sub-states of the \"Opening\" state.\r\n"
                      "\ttimeCameraTrigAdv            Time period by which the camera trigger is advanced\r\n"
                      "\t                             before the shutter's \"Opening\" state starts. This\r\n"
                      "\t                             value must be smaller than <timeShutterStayClosed>.\r\n"
```

```
                        "\ttimeCameraRecorords          Time period for which camera trigger will be active.\r\n"
                        "\ttimeOfExperiment             Total time of one experiment.\r\n"
                        "\r\n"
                        "Example:\r\n"
                        "    >cycleTiming 200000 1000 500000 1000000\r\n"
                        "\r\n"
                        "For the exaple above, the shutter will stay  in \"Closed\" state for 200ms.\r\n"
                        "The camera trigger will be active 1ms befor the shutter goes into \"Opening\"\r\n"
                        "state. This trigger will last for 500ms. And, the complete experiment will\r\n"
                        "last for 1s.\r\n";

    // Print out the documentation
    putsUSB(doc);
}
```

G.2.3 helpDocs.h

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#ifndef __HELPDOCS_DEFINED__
#define __HELPDOCS_DEFINED__

#include "serialCOM.h"

void help(const char *command);
void help_about();
void help_shutterForce();
void help_openingTiming();
void help_closingTiming();
void help_cycleTiming();
void help_start();
void help_stop();
void help_status();
void help_reboot();

#endif
```

G.2.4 serialCOM.cpp

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#include "serialCOM.h"

// Blocking function, if timeput_us = 0, that returns with a character when available.
// Otherwise the number would be negative if no character is received within timeout.
int getcUSB(uint32_t timeout_us)
{
    int c;
    if (timeout_us)
        c = getchar_timeout_us(timeout_us);
    else
        while ((c = getchar_timeout_us(0)) < 0);
    return c;
}

// Blocking function, if timeput_us = 0, that reads a string when available.
// The timeout_us value is used for each character.
bool getsUSB(struct serialInput *userInput, uint32_t timeout_us)
{
    uint8_t count = 0;
    int c;

    while (count < MAX_CHARS)
    {
        c = getcUSB(timeout_us);
        if (c == 8 || c == 127)
        {
            if (count == 0)
                continue;
            count--;
        }
        else if (c == 10 || c == 13) // Putty generally sends '\r' on Enter key
        {
            userInput->str[count] = '\0'; // Add Null character
            c = getcUSB(1000); // Flush out the expected '\n' character if received within 1ms
            break;
        }
        else if (c > 31 && c < 127)
            userInput->str[count++] = c;
        else if (!timeout_us)
            continue;
        else
```

```
            return 0;
    }
    parseString(userInput);
    return 1;
}

uint32_t intPow(uint32_t x, uint32_t p)
{
    uint32_t i = 1;
    for (uint32_t j = 0; j < p; j++)
        i *= x;
    return i;
}

// void itoa(uint32_t iVal, char *str, uint8_t strLength)
// {
//     uint8_t p = 255;
//     while (iVal / intPow(10, ++p) != 0)
//         ;
//     if (p)
//     {
//         str[p] = '\0';
//         for (uint8_t i = 1; i < strLength - 1; ++i)
//         {
//             str[p - i] = '0' + (iVal % 10);
//             iVal /= 10;
//             if (i == p)
//             {
//                 strLength = p;
//                 break;
//             }
//         }
//     }
//     else
//     {
//         str[0] = '0';
//         str[1] = '\0';
//     }
//     str[strLength] = '\0';
// }

void getPercentage(uint32_t iVal, char *str, uint8_t strLength)
{
    str[4] = iVal % 10 + '0';
    iVal /= 10;
    str[3] = iVal % 10 + '0';
    iVal /= 10;
    str[2] = '.';
    str[1] = iVal % 10 + '0';
    iVal /= 10;
    str[0] = iVal % 10 + '0';
```

```
    iVal /= 10;
    str[5] = '\0';
}
```

G.2.5 serialCOM.h

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#ifndef __SERIALCOM_DEFINED__
#define __SERIALCOM_DEFINED__

#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "serialInput.h"

// Define tokens used for communication
#define sndToken "115104117116116101114067111110116114111108108101114\r\n" // Each 3 digits
↪  converted to chars results in 'shutterController'
#define recToken "116119101101122101114083111102116119097114101"           // Each 3 digits
↪  converted to chars results in 'tweezerSoftware'
#define endRspToken "101110100067109100082115112\r\n"                        // Each 3 digits
↪  converted to chars results in 'endCmdRsp'

#define putcUSB(c) printf("%c", c) // Blocking function that writes a character
#define putsUSB(str_ptr) printf("%s", str_ptr) // Blocking function that writes a string

int getcUSB(uint32_t timeout_us = 0);
bool getsUSB(struct serialInput *userInput, uint32_t timeout_us = 0);
// void itoa(uint32_t iVal, char *str, uint8_t strLength);
void getPercentage(uint32_t iVal, char *str, uint8_t strLength);

#endif
```

G.2.6 pico_sdk_import.cmake

```
# This is a copy of <PICO_SDK_PATH>/external/pico_sdk_import.cmake

# This can be dropped into an external project to help locate this SDK
# It should be include()ed prior to project()

if (DEFINED ENV{PICO_SDK_PATH} AND (NOT PICO_SDK_PATH))
```

```
    set(PICO_SDK_PATH $ENV{PICO_SDK_PATH})
    message("Using PICO_SDK_PATH from environment ('${PICO_SDK_PATH}')")
endif ()

if (DEFINED ENV{PICO_SDK_FETCH_FROM_GIT} AND (NOT PICO_SDK_FETCH_FROM_GIT))
    set(PICO_SDK_FETCH_FROM_GIT $ENV{PICO_SDK_FETCH_FROM_GIT})
    message("Using PICO_SDK_FETCH_FROM_GIT from environment ('${PICO_SDK_FETCH_FROM_GIT}')")
endif ()

if (DEFINED ENV{PICO_SDK_FETCH_FROM_GIT_PATH} AND (NOT PICO_SDK_FETCH_FROM_GIT_PATH))
    set(PICO_SDK_FETCH_FROM_GIT_PATH $ENV{PICO_SDK_FETCH_FROM_GIT_PATH})
    message("Using PICO_SDK_FETCH_FROM_GIT_PATH from environment ('${PICO_SDK_FETCH_FROM_GIT_PATH}')")
endif ()

set(PICO_SDK_PATH "${PICO_SDK_PATH}" CACHE PATH "Path to the Raspberry Pi Pico SDK")
set(PICO_SDK_FETCH_FROM_GIT "${PICO_SDK_FETCH_FROM_GIT}" CACHE BOOL "Set to ON to fetch copy of SDK from git if not otherwise locatable")
set(PICO_SDK_FETCH_FROM_GIT_PATH "${PICO_SDK_FETCH_FROM_GIT_PATH}" CACHE FILEPATH "location to download SDK")

if (NOT PICO_SDK_PATH)
    if (PICO_SDK_FETCH_FROM_GIT)
        include(FetchContent)
        set(FETCHCONTENT_BASE_DIR_SAVE ${FETCHCONTENT_BASE_DIR})
        if (PICO_SDK_FETCH_FROM_GIT_PATH)
            get_filename_component(FETCHCONTENT_BASE_DIR "${PICO_SDK_FETCH_FROM_GIT_PATH}" REALPATH BASE_DIR "${CMAKE_SOURCE_DIR}")
        endif ()
        FetchContent_Declare(
                pico_sdk
                GIT_REPOSITORY https://github.com/raspberrypi/pico-sdk
                GIT_TAG master
        )
        if (NOT pico_sdk)
            message("Downloading Raspberry Pi Pico SDK")
            FetchContent_Populate(pico_sdk)
            set(PICO_SDK_PATH ${pico_sdk_SOURCE_DIR})
        endif ()
        set(FETCHCONTENT_BASE_DIR ${FETCHCONTENT_BASE_DIR_SAVE})
    else ()
        message(FATAL_ERROR
                "SDK location was not specified. Please set PICO_SDK_PATH or set PICO_SDK_FETCH_FROM_GIT to on to fetch from git."
                )
    endif ()
endif ()

get_filename_component(PICO_SDK_PATH "${PICO_SDK_PATH}" REALPATH BASE_DIR "${CMAKE_BINARY_DIR}")
```

```
if (NOT EXISTS ${PICO_SDK_PATH})
    message(FATAL_ERROR "Directory '${PICO_SDK_PATH}' not found")
endif ()

set(PICO_SDK_INIT_CMAKE_FILE ${PICO_SDK_PATH}/pico_sdk_init.cmake)
if (NOT EXISTS ${PICO_SDK_INIT_CMAKE_FILE})
    message(FATAL_ERROR "Directory '${PICO_SDK_PATH}' does not appear to contain the Raspberry Pi Pico SDK")
endif ()

set(PICO_SDK_PATH ${PICO_SDK_PATH} CACHE PATH "Path to the Raspberry Pi Pico SDK" FORCE)

include(${PICO_SDK_INIT_CMAKE_FILE})
```

G.2.7 serialInput.cpp

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#include "serialInput.h"

void parseString(struct serialInput *userInput)
{
    bool prevCharState = 0; // Previous Character state. 0 = Character not useful, 1 = Character is useful
    userInput->argCount = 0;
    uint8_t length = strlen(userInput->str);
    for (uint8_t i = 0; i < length; ++i)
    {
        char c = userInput->str[i];
        if ((c == 43 || c == 46 || c == 38) || (c > 47 && c < 58) || (c > 64 && c < 91) || (c > 96 && c < 123))
        {
            if (prevCharState == 0)
            {
                if (userInput->argCount == MAX_FIELDS)
                {
                    putsUSB("\r\n> Number of arguments entered is more than maximum limit.");
                    putsUSB("\r\n> All the arguments after maximum limit will be ignored.");
                    break;
                }
                userInput->pos[userInput->argCount++] = i;
                prevCharState = 1;
```

```c
            }
            // if (c>64 && c<91)
            // {
            //         userInput->str[i] += 32;
            // }
        }
        else
        {
            prevCharState = 0;
            userInput->str[i] = '\0';
        }
    }
}

const char *getArgString(struct serialInput *userInput, uint8_t argNumber)
{
    if (argNumber + 1 <= userInput->argCount)
        return userInput->str + userInput->pos[argNumber];
    else
        return NULL;
}
```

G.2.8 serialInput.h

```c
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#ifndef __SERIAL_INPUT__
#define __SERIAL_INPUT__

#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include "serialCOM.h"

#define MAX_CHARS 80
#define MAX_FIELDS 6

struct serialInput
{
    char str[MAX_CHARS + 1];
    uint8_t pos[MAX_FIELDS], argCount;
};

void parseString(struct serialInput *userInput);
```

```
const char *getArgString(struct serialInput *userInput, uint8_t argNumber);

#endif
```

G.2.9 shutter.cpp

```
//  Author: Vatsal Asitkumar Joshi
//  Date: May 23rd, 2022
//  This code is a set of functions to work with USB serial communication.
//
//  "If you are done writing the code, now is a good time to debug it."
//

#include <stdio.h>
#include <stdlib.h>
#include "pico/stdlib.h"
#include "hardware/pwm.h"
#include "hardware/timer.h"
#include "hardware/watchdog.h"
#include "serialCOM.h"
#include "helpDocs.h"

// Define alarm numbers and IRQ vector numbers for shutter and camera timing
#define SHTR_ALARM_NUM 0
#define CMRA_ALARM_NUM 1

// Define the necessary pins
const uint blueLED = 14;
const uint shutterInA = 16;
const uint shutterInB = 17;
const uint camTrigger = 9;
const uint centrifuge = 2;

// Shutter PWM specific definitions
#define SYSTEM_FREQ 125000000                      // RP2040  clock frequency 125MHz
#define SHTR_PWM_FREQ 20000                        // Desired PWM output frequency 20kHz
#define SHTR_PWM_TOP (SYSTEM_FREQ / SHTR_PWM_FREQ) // Value that TOP register should have to get desired PWM frequency

// Blue LED PWM specific definitions
#define BLED_PWM_FREQ 125000                       // Desired PWM output frequency 125kHz
#define BLED_PWM_TOP (SYSTEM_FREQ / BLED_PWM_FREQ) // Value that TOP register should have to get desired PWM frequency

// Structures to hold shutter timings
struct motion
{
    uint start_us; // Time for which power will be applied in one direction during motion
    uint delay_us; // Time for which power will be off during motion
```

```
    uint stop_us;  // Time for which power will be in the opposite direction to stop the motion
};
struct _cycle
{
    struct motion opening; // Shutter opening motion timings
    struct motion closing; // Shutter closing motion timings
    uint closed_us;        // Time for which shutter stay closed
    uint camTrigAdv_us;    // Camera trigger applied befor this many microseconds of shutter start opening
    uint camTrigLen_us;    // Time period for which the camera will be recording
    // Length of one cycle = closing + closed_us + opening + camTrigLen_us - camTrigAdv_us
} shutter = {4500, 800, 2500, 4500, 800, 2500, 200000, 1000, 500000};
struct _pwm
{
    uint movingDutyCycle;                  // PWM CC register value while the shutter is moving
    uint hldingDutyCycle;                  // PMW CC register value while the shutter is not moving
} pwm = {SHTR_PWM_TOP + 1, SHTR_PWM_TOP * 10 / 100}; // Default values: movingDutyCycle = 100%, hldingDutyCycle = 1%

// Define necessary global variables
struct serialInput usbSerialIn;     // Container to hold user input
volatile bool expInProcess = false; // States whether some experiments are currently happening or not
volatile bool camRecording = false; // Status whether camera is currently recording or not
uint expLen_us = shutter.closing.start_us + shutter.closing.delay_us + shutter.closing.stop_us
               + shutter.closed_us
               + shutter.opening.start_us + shutter.opening.delay_us + shutter.opening.stop_us
               + shutter.camTrigLen_us - shutter.camTrigAdv_us;

// Define different states the shutter could be in
enum shutterState
{
    opened,
    closing_start,
    closing_delay,
    closing_stop,
    closed,
    opening_start,
    opening_delay,
    opening_stop
};

void shutterISR(uint alarmNum)
{
    static enum shutterState lastState = opened;    // Assume that initially the shutter is open
```

```
    static absolute_time_t expStartTime;            // Time at which the experiment was started

    // Get current time
    absolute_time_t t = get_absolute_time();

    // State that experiments are hapenning
    expInProcess = true;

    // Do stuff based on what the last state of the shutter was
    switch (lastState)
    {
    case opened: // If the shutter was opened then start closing it
        // Make INA = 100% and INB = 0%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), pwm.movingDutyCycle);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call this ISR after shutter closing start
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.closing.start_us));

        // Save experiment start time
        expStartTime = t;

        // Make shutter state closing_start
        lastState = closing_start;
        break;
    case closing_start: // If the shutter has started closing then cut off the power
        // Make INA = 0% and INB = 0%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call this ISR after shutter closing delay
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.closing.delay_us));

        // Make shutter state closing_delay
        lastState = closing_delay;
        break;
    case closing_delay: // If the shutter was in the closing delay then supply power in opposite direction to stop the motion
        // Make INA = 0% and INB = 100%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), pwm.movingDutyCycle);

```

```c
        // Call this ISR after shutter closing stop
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.closing.stop_us));

        // Make shutter state closing_stop
        lastState = closing_stop;
        break;
    case closing_stop: // If the shutter was stopping the closing motion then keep it in the closed position
        // Make INA = 1% and INB = 0
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), pwm.hldingDutyCycle);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call this ISR after the time shutter should stay closed
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.closed_us));

        // Call cameraISR slightly before shutter starts to open
        hardware_alarm_set_target(CMRA_ALARM_NUM, delayed_by_us(t, shutter.closed_us - shutter.camTrigAdv_us));

        // Make shutter state closed
        lastState = closed;
        break;
    case closed: // If the shutter was closed then start opening it
        // Make INA = 0% and INB = 100%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), pwm.movingDutyCycle);

        // Call this ISR after timeShutterMoving
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.opening.start_us));

        // Make shutter state opening_start
        lastState = opening_start;
        break;
    case opening_start: // If the shutter has started opening then cut off the power
        // Make INA = 0% and INB = 0%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call this ISR after timeShutterMoving
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.opening.delay_us));

        // Make shutter state opening_delay
```

```
        lastState = opening_delay;
        break;
    case opening_delay: // If the shutter was in the opening delay then supply power in opposite direction to stop the motion
        // Make INA = 100% and INB = 0%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), pwm.movingDutyCycle);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call this ISR after timeShutterMoving
        hardware_alarm_set_target(SHTR_ALARM_NUM, delayed_by_us(t, shutter.opening.stop_us));

        // Make shutter state opening_stop
        lastState = opening_stop;
        break;
    default: // If the shutter was stopping the opening motion then cut off the power
        // Make INA = 0% and INB = 1%
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        if (shutter.closed_us + shutter.opening.start_us + shutter.opening.delay_us + shutter.opening.stop_us >= shutter.closed_us - shutter.camTrigAdv_us + shutter.camTrigLen_us)
            expInProcess = false; // State that the experiment is done if the camera trigger end happens before shutter is openned

        // Make shutter state opened
        lastState = opened;
        break;
    }
}

void cameraISR(uint alarmNum)
{
    // Get current time
    absolute_time_t t = get_absolute_time();

    if (camRecording) // If camera was recording previously then
    {
        gpio_put(camTrigger, 0); // Make camera trigger pin = 0
        camRecording = false;    // State that camera has stopped recording
        if (shutter.closed_us + shutter.opening.start_us + shutter.opening.delay_us + shutter.opening.stop_us <= shutter.closed_us - shutter.camTrigAdv_us + shutter.camTrigLen_us)
            expInProcess = false;    // State that the experiment is done if the camera trigger end happens after shutter is openned
    }
```

```c
    else // If camera was not recording previously then start recording
    {
        gpio_put(camTrigger, 1); // Make camera trigger pin = 1
        camRecording = true; // State that camera has stopped recording
        hardware_alarm_set_target(CMRA_ALARM_NUM, delayed_by_us(t, shutter.camTrigLen_us)); // Call this ISR when camera has to stop recording
    }
}

// Function that initializes all the pins and hardware
void setup()
{
    stdio_init_all();

    // Initialize GPIOs for shutter as output
    gpio_init(shutterInA);
    gpio_set_dir(shutterInA, GPIO_OUT);
    // gpio_set_slew_rate(shutterInA, GPIO_SLEW_RATE_FAST);
    // gpio_set_drive_strength(shutterInA, GPIO_DRIVE_STRENGTH_8MA);
    gpio_init(shutterInB);
    gpio_set_dir(shutterInB, GPIO_OUT);
    // gpio_set_slew_rate(shutterInB, GPIO_SLEW_RATE_FAST);
    // gpio_set_drive_strength(shutterInB, GPIO_DRIVE_STRENGTH_8MA);

    // Initialize GPIOs for shutter as PWM
    gpio_set_function(shutterInA, GPIO_FUNC_PWM);
    gpio_set_function(shutterInB, GPIO_FUNC_PWM);
    pwm_set_wrap(pwm_gpio_to_slice_num(shutterInA), SHTR_PWM_TOP); // Set the counter wrap value based on the desired PWM frequency
    pwm_set_wrap(pwm_gpio_to_slice_num(shutterInB), SHTR_PWM_TOP); // Set the counter wrap value based on the desired PWM frequency
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0); // Chan A (GPIO 0) counter compare value (Decides duty cycle)
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0); // Chan B (GPIO 1) counter compare value (Decides duty cycle)
    pwm_set_enabled(pwm_gpio_to_slice_num(shutterInA), true); // Enable the PWM counter for InA
    pwm_set_enabled(pwm_gpio_to_slice_num(shutterInB), true); // Enable the PWM counter for InA

    // Initialize GPIO for blue LED
    gpio_set_function(blueLED, GPIO_FUNC_PWM);
    pwm_set_wrap(pwm_gpio_to_slice_num(blueLED), BLED_PWM_TOP);                      // Set the counter wrap value based on the desired PWM frequency
    pwm_set_chan_level(pwm_gpio_to_slice_num(blueLED), pwm_gpio_to_channel(blueLED), 0); // Chan A (GPIO 14) counter compare value (Decides duty cycle)
    pwm_set_enabled(pwm_gpio_to_slice_num(blueLED), true);                           // Enable the PWM counter for Blue LED

```

```
    // Initialize GPIO for camera as output
    gpio_init(camTrigger);
    gpio_set_dir(camTrigger, GPIO_OUT);
    gpio_put(shutterInB, 0);

    // Initialize GPIO for centrifuge as output
    gpio_init(centrifuge);
    gpio_set_dir(centrifuge, GPIO_OUT);

    // Initialize GPIO for centrifuge as PWM
    gpio_set_function(centrifuge, GPIO_FUNC_PWM);
    pwm_set_clkdiv_int_frac(pwm_gpio_to_slice_num(centrifuge), 125, 0); // Set the clock divider to 125 to get 1us reference
    pwm_set_wrap(pwm_gpio_to_slice_num(centrifuge), 10000); // Set the counter wrap value to 10000 to get 10ms cycle time
    pwm_set_gpio_level(centrifuge, 1000); // Set default pulse width to 1ms
    pwm_set_enabled(pwm_gpio_to_slice_num(centrifuge), true); // Enable the pwm output

    // Make sure the shutter is in open position by setting shutterInA = 0% and shutterInB = movingDutyCycle/2% duty cycle
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), pwm.movingDutyCycle/2);
    sleep_ms(500);
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
    pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

    // Make sure camera trigger is off
    gpio_put(camTrigger, 0);

    // Set up ISRs and enable IRQs for shutter and camera alarms
    hardware_alarm_claim(SHTR_ALARM_NUM);                       // Claim an alarm for shutter and enable IRQ
    hardware_alarm_set_callback(SHTR_ALARM_NUM, shutterISR);    // Define the callback function for shutter alarm
    hardware_alarm_claim(CMRA_ALARM_NUM);                       // Claim an alarm for camera trigger and enable IRQ
    hardware_alarm_set_callback(CMRA_ALARM_NUM, cameraISR);    // Define the callback function for camera trigger

    // Wait for the USB serial connection to be activated
    while (!stdio_usb_connected());

    // Keep sending the secret message untill a correct response is received
    char s[80];
    bool strRcv = 0;
    do
    {
```

```
        putsUSB(sndToken); // Send a token for handshake
        strRcv = getsUSB(&usbSerialIn, 200e3);
        strcpy(s, getArgString(&usbSerialIn, 0));
    } while (!strRcv || strcmp(s, recToken)); // Check if the received token is correct or not
}

void loop()
{
    // Read user input
    getsUSB(&usbSerialIn);
    const char *command = getArgString(&usbSerialIn, 0);

    // Execute specific command based on user input
    if (!strcmp(command, "help") && usbSerialIn.argCount < 3)
    {
        // Provide general discussion on how the shutter and this program works
        help(getArgString(&usbSerialIn, 1));
    }
    else if (!strcmp(command, "shutterForce") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        putsUSB("Shutter force when it is moving    = ");
        itoa(pwm.movingDutyCycle * 100 / SHTR_PWM_TOP, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" %%\r\n");
        putsUSB("Shutter force when it is closed    = ");
        itoa(pwm.hldingDutyCycle * 100 / SHTR_PWM_TOP, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" %%\r\n");
    }
    else if (!strcmp(command, "shutterForce") && usbSerialIn.argCount == 3)
    {
        uint movingForce = atoi(getArgString(&usbSerialIn, 1));
        uint hldingForce = atoi(getArgString(&usbSerialIn, 2));

        if (movingForce < 101 && hldingForce < 101)
        {
            pwm.movingDutyCycle = movingForce == 100 ? SHTR_PWM_TOP + 1 : SHTR_PWM_TOP * movingForce / 100;
            pwm.hldingDutyCycle = hldingForce == 100 ? SHTR_PWM_TOP + 1 : SHTR_PWM_TOP * hldingForce / 100;

            putsUSB("Warning: Choose the shutter moving and holding force carefully.\r\nContinuous high force can damage the shutter.\r\n");
        }
        else
        {
            putsUSB("The shutter force values have to be between 0%% and 100%% inclusive.\r\n");
        }
```

```
    }
    else if (!strcmp(command, "openingTiming") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        putsUSB("Power supplied for ");
        itoa(shutter.opening.start_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us to start opening the shutter.\r\n");
        putsUSB("Power cut off for ");
        itoa(shutter.opening.delay_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us. \r\n");
        putsUSB("Power supplied in opposite direction for ");
        itoa(shutter.opening.stop_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us to stop shutter motion.\r\n");
    }
    else if (!strcmp(command, "openingTiming") && usbSerialIn.argCount == 4)
    {
        uint start = atoi(getArgString(&usbSerialIn, 1));
        uint delay = atoi(getArgString(&usbSerialIn, 2));
        uint stop = atoi(getArgString(&usbSerialIn, 3));

        if (start && delay && stop)
        {
            shutter.opening.start_us = start;
            shutter.opening.delay_us = delay;
            shutter.opening.stop_us = stop;
        }
        else
        {
            putsUSB("Motion timing values cannot be 0.\r\n");
        }
    }
    else if (!strcmp(command, "closingTiming") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        putsUSB("Power supplied for ");
        itoa(shutter.closing.start_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us to start closing the shutter.\r\n");
        putsUSB("Power cut off for ");
        itoa(shutter.closing.delay_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us. \r\n");
        putsUSB("Power supplied in opposite direction for ");
        itoa(shutter.closing.stop_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us to stop shutter motion.\r\n");
    }
    else if (!strcmp(command, "closingTiming") && usbSerialIn.argCount == 4)
```

```
    {
        uint start = atoi(getArgString(&usbSerialIn, 1));
        uint delay = atoi(getArgString(&usbSerialIn, 2));
        uint stop = atoi(getArgString(&usbSerialIn, 3));

        if (start && delay && stop)
        {
            shutter.closing.start_us = start;
            shutter.closing.delay_us = delay;
            shutter.closing.stop_us = stop;
        }
        else
        {
            putsUSB("Motion timing values cannot be 0.\r\n");
        }
    }
    else if (!strcmp(command, "cycleTiming") && usbSerialIn.argCount == 4)
    {
        // Update the values of experiment parameters
        shutter.closed_us = atoi(getArgString(&usbSerialIn, 1));
        shutter.camTrigAdv_us = atoi(getArgString(&usbSerialIn, 2));
        shutter.camTrigLen_us = atoi(getArgString(&usbSerialIn, 3));

        if (shutter.camTrigAdv_us >= shutter.closed_us)
        {
            putsUSB("The camera trigger can be advanced at max till the shutter closes. Make sure that\r\n");
            putsUSB("   timeCameraTrigAdv < timeShutterStayClosed\r\n");
            putsUSB("For now timeCameraTrigAdv is set to 0 us.\r\n");
            shutter.camTrigAdv_us = 0; // Set timeCameraTrigAdv to 0
        }

        expLen_us = shutter.closing.start_us + shutter.closing.delay_us + shutter.closing.stop_us + shutter.closed_us + shutter.opening.start_us + shutter.opening.delay_us + shutter.opening.stop_us;
        if (shutter.closed_us + shutter.opening.start_us + shutter.opening.delay_us + shutter.opening.stop_us <= shutter.closed_us - shutter.camTrigAdv_us + shutter.camTrigLen_us)
            expLen_us += shutter.camTrigLen_us - shutter.camTrigAdv_us;
    }
    else if (!strcmp(command, "cycleTiming") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        putsUSB("Shutter will stay closed for ");
        itoa(shutter.closed_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us.\r\n");
        putsUSB("Camera will be triggered ");
        itoa(shutter.camTrigAdv_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us before shutter starts opening. \r\n");
```

```
        putsUSB("Camera will record for ");
        itoa(shutter.camTrigLen_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us.\r\n");
        putsUSB("One complete experiment will be ");
        itoa(expLen_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB(" us long.\r\n");
    }
    else if (!strcmp(command, "centrifuge") && usbSerialIn.argCount == 2)
    {
        // Update the centrifuge pulse-width
        int cc = atoi(getArgString(&usbSerialIn, 1));
        if (cc >= 0 && cc <= 9)
            pwm_set_gpio_level(centrifuge, cc * 100 + 1000);
        else
            pwm_set_gpio_level(centrifuge, 1000);
    }
    else if (!strcmp(command, "led") && usbSerialIn.argCount == 2)
    {
        // Update the blue LED duty cycle
        int cc = atoi(getArgString(&usbSerialIn, 1));
        if (cc >= 0 && cc <= 100)
            pwm_set_gpio_level(blueLED, BLED_PWM_TOP * cc / 100);
        else
            pwm_set_gpio_level(blueLED, 0);
    }
    else if (!strcmp(command, "shtrOpen") && usbSerialIn.argCount == 1)
    {
        // Open the shutter
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), pwm.movingDutyCycle/2);
        sleep_ms(500);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);
    }
    else if (!strcmp(command, "shtrClose") && usbSerialIn.argCount == 1)
    {
        // Close the shutter
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), pwm.movingDutyCycle/2);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);
        sleep_ms(500);
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
```

```
        pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);
    }
    else if (!strcmp(command, "++expLen") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        itoa(expLen_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB("\r\n");
    }
    else if (!strcmp(command, "++camTrigLen") && usbSerialIn.argCount == 1)
    {
        char outNumber[11];
        itoa(shutter.camTrigLen_us, outNumber, 10);
        putsUSB(outNumber);
        putsUSB("\r\n");
    }
    else if (!strcmp(command, "++start") && usbSerialIn.argCount < 3)
    {
        // // Make sure that the shutter is in open state
        // pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        // pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), pwm.hldingDutyCycle);
        // sleep_ms(500);
        // pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInA), pwm_gpio_to_channel(shutterInA), 0);
        // pwm_set_chan_level(pwm_gpio_to_slice_num(shutterInB), pwm_gpio_to_channel(shutterInB), 0);

        // Call the shutter alarm function after 1ms to initiate the experiment
        hardware_alarm_set_target(SHTR_ALARM_NUM, make_timeout_time_ms(1));
        // shutterISR(SHTR_ALARM_NUM);

        // Block all the other operations untill the experiment ends
        while (expInProcess);
    }
    else if (!strcmp(command, "++reboot") && usbSerialIn.argCount)
    {
        putsUSB("The shutter controller is set to reboot.");
        putsUSB("\r\n");
        watchdog_reboot(0, 0, 100);
    }
    else if (!usbSerialIn.argCount)
    {
    }
    else
    {
        putsUSB("\tInvalid Command: ");
        putsUSB(command);
        putsUSB("\r\n");
```

```
        putsUSB("\tType \"help\" for list of commands.\r\n");
        putsUSB("\tType \"help command\" for documentation of a specific command.\r\n");
    }
    // Send CR/LF pair to be on safe side
    putsUSB("\r\n");

    // Output a token to indicate end of command response
    putsUSB(endRspToken);
}

int main()
{
    setup();

    while (true)
    {
        loop();
    }
}
```

APPENDIX H

Code Listings for the Host-PC software

The code provided here relies on many other pieces of software listed below:

- `cli`: A library to create command line interface. It is available at https://github.com/daniele77/cli.
- `DCAM-API`: Primarily provides a driver for the frame grabber card (FireBird 1XCLD-2PE8) from Active Silicon. It also has other tools that can be used for testing and configuration purposes, like `ExCap4` and `DCAM Configurator`. It can be downloaded from https://dcam-api.com/downloads/
- `DCAM-SDK`: This is the software development kit provided by Hamamatsu. It provides important functions and headers to properly communicate with the camera. It can be downloaded from https://dcam-api.com/dcam-sdk-login/.
- `glew`, `glfw` and `glm`: These are OpenGL libraries that facilitate the live view functionality for the camera. Only certain files from each library are required, take a look at the directory tree and `CMakeLists.txt` for help. Following are the download links for each
  - `glew`: https://sourceforge.net/projects/glew/files/glew/2.1.0/glew-2.1.0-win32.zip/download
  - `glfw`: https://www.glfw.org/download
  - `glm`: https://github.com/g-truc/glm
- `serialib`: This library facilitates the communication between the host pc and the microcontroller. It can be downloaded from https://github.com/imabot2/serialib

H.1 Directory tree

```
/
├── cli/
├── dcamMisc/ ....... Modified versions of some files in dcamsdk4/samples/cpp/misc
│   ├── common.cpp
│   ├── common.h
│   └── console4.h
├── dcamsdk4/ ....... Everything else is the same except the following
│   └── inc/
│       └── dcamapi4.h ....... Modified version of dcamsdk4/inc/dcamapi4.h
├── glew-2.1.0/
├── glfw-3.3.8.bin.WIN64/
├── glm/
├── serialib/
├── camProp.cpp ....... Get and Set Camera Properties
├── camProp.h ....... Header file for camProp.cpp
├── camRec.cpp ....... Records images from camera
├── camRec.h ....... Header file for camRec.cpp
├── camSoft.cpp ....... Main code of the host pc software
├── camSoft.h ....... Header file for camSoft.cpp
├── CMakeLists.txt ....... CMake configuration file
├── condExps.cpp ....... Automates conducting experiments
├── condExps.h ....... Header file for condExps.cpp
├── liveCap.cpp ....... Provides live camera stream
├── liveCap.h ....... Header file for liveCap.cpp
├── shtrCtrl.cpp ....... Communicates with the microcontroller
└── shtrCtrl.h ....... Header file for shtrCtrl.cpp
```

H.2 Code Listings

H.2.1 `common.cpp`

Make the following changes to the original file, remove the lines that are highlighted in red and add the lines that are highlighted in green.

```
// console/misc/common.cpp
//

#include    "console4.h"
#include    "common.h"

#include    <stdarg.h>
```

```
#ifndef ASSERT
#define ASSERT(c)
#endif

// ----------------------------------------------------------------
```

H.2.2 `common.h`

Make the following changes to the original file, remove the lines that are highlighted in red and add the lines that are highlighted in green.

```
// console/misc/common.h
//

#include    "console4.h"
#include    <stdarg.h>

#ifndef ASSERT
#define ASSERT(c)
#endif

void dcamcon_show_dcamerr( HDCAM hdcam, DCAMERR errid, const char* apiname, const char* fmt=0, ...  );
```

H.2.3 `console4.h`

Make the following changes to the original file, remove the lines that are highlighted in red and add the lines that are highlighted in green.

```
#if defined( LINUX )
#include            "dcamapi4.h"
#include            "dcamprop.h"
#else
#include            "../../../../inc/dcamapi4.h"
#include            "../dcamsdk4/inc/dcamapi4.h"
#include            "../../../../inc/dcamprop.h"
#include            "../dcamsdk4/inc/dcamprop.h"
#endif

#if defined(_WIN64)
#pragma comment(lib,"../../../../lib/win64/dcamapi.lib")
#elif defined(WIN32)
#pragma comment(lib,"../../../../lib/win32/dcamapi.lib")
#endif
```

```

#endif // _NO_DCAMAPI
```

H.2.4 dcamapi4.h

Make the following changes to the original file, remove the lines that are highlighted in red and add the lines that are highlighted in green.

```
inline int failed( DCAMERR err )
{
    return int(err) < 0;
}

#endif

#if (defined(_MSC_VER)&&defined(_LINK_DCAMAPI_LIB))
#pragma comment(lib, "dcamapi.lib")
#endif

#pragma pack()

#define _INCLUDE_DCAMAPI4_H_
#endif
```

H.2.5 camProp.cpp

```
#include "camProp.h"

// Define global variables
std::vector<camPropInfo> camProps; // Array to store camera properties

camPropInfo get_camPropInfo(HDCAM hdcam, int32 propID)
{
        camPropInfo propInfo;
        DCAMERR err;

        propInfo.propID = propID;
        memset(&propInfo.propAttr, 0, sizeof(propInfo.propAttr));
        propInfo.propAttr.cbSize = sizeof(propInfo.propAttr);
        propInfo.propAttr.iProp = propID;

        dcamprop_getname(hdcam, propID, propInfo.propName, propNameSize);        // get property name
        err = dcamprop_getattr(hdcam, &propInfo.propAttr); // get property attribute

        // Collect prop attribute info
```

```
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASCHANNEL)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "HASCHANNEL");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_AUTOROUNDING)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "AUTOROUNDING");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_STEPPING_INCONSISTENT)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "STEPPING_INCONSISTENT");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_DATASTREAM)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "DATASTREAM");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASRATIO)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "HASRATIO");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_VOLATILE)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "VOLATILE");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_WRITABLE)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "WRITABLE");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_READABLE)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "READABLE");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_ACCESSREADY)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "ACCESSREADY");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_ACCESSBUSY)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "ACCESSBUSY");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR_EFFECTIVE)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "EFFECTIVE");

        // Collect prop attribute2 info
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR2_ARRAYBASE)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "ARRAYBASE");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR2_ARRAYELEMENT)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "ARRAYELEMENT");
        if (propInfo.propAttr.attribute & DCAMPROP_ATTR2_INITIALIZEIMPROPER)
                strcpy_s(propInfo.attrNames[propInfo.nAttr++], attrNameSize, "INITIALIZEIMPROPER");

        // Get the datatype of the property
        switch (propInfo.propAttr.attribute & DCAMPROP_TYPE_MASK)
        {
        case DCAMPROP_TYPE_MODE:        strcpy_s(propInfo.dataType, datatypeNameSize, "MODE");        break;
        case DCAMPROP_TYPE_LONG:        strcpy_s(propInfo.dataType, datatypeNameSize, "LONG");        break;
        case DCAMPROP_TYPE_REAL:        strcpy_s(propInfo.dataType, datatypeNameSize, "REAL");        break;
        default:                                        strcpy_s(propInfo.dataType, datatypeNameSize, "NONE");        break;
        }

        // Extract the range of property values
```

```
        propInfo.min = (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASRANGE) ? propInfo.propAttr.valuemin : std::numeric_limits<double>::quiet_NaN();
        propInfo.max = (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASRANGE) ? propInfo.propAttr.valuemax : std::numeric_limits<double>::quiet_NaN();

        // Extract the step size of property value
        propInfo.step = (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASSTEP) ? propInfo.propAttr.valuestep : std::numeric_limits<double>::quiet_NaN();

        // Extract the default value of the property
        propInfo.defaultVal = (propInfo.propAttr.attribute & DCAMPROP_ATTR_HASDEFAULT) ? propInfo.propAttr.valuedefault : std::numeric_limits<double>::quiet_NaN();

        // Extract the current value of the property
        dcamprop_getvalue(hdcam, propID, &propInfo.currentVal);

        // Get all possible values if the datatype is MODE
        if ((propInfo.propAttr.attribute & DCAMPROP_TYPE_MASK) == DCAMPROP_TYPE_MODE) // List possible values for the property
        {
                double lastValue = propInfo.min;
                ++propInfo.nSuppVals;
                do {
                        DCAMPROP_VALUETEXT pvt;
                        memset(&pvt, 0, sizeof(pvt));
                        pvt.cbSize = sizeof(pvt);
                        pvt.iProp = propID;
                        pvt.value = lastValue;
                        propInfo.suppVals[propInfo.nSuppVals] = int32(lastValue);
                        pvt.text = propInfo.suppValNames[propInfo.nSuppVals++];
                        pvt.textbytes = suppValNameSize;
                        dcamprop_getvaluetext(hdcam, &pvt);
                } while (!failed(dcamprop_queryvalue(hdcam, propID, &lastValue, DCAMPROP_OPTION_NEXT)));
        }

        // Get the unit of property value
        switch (propInfo.propAttr.iUnit)
        {
        case DCAMPROP_UNIT_SECOND:                        strcpy_s(propInfo.unit, unitNameSize, "SECOND");                        break;
        case DCAMPROP_UNIT_CELSIUS:                       strcpy_s(propInfo.unit, unitNameSize, "CELSIUS");                       break;
        case DCAMPROP_UNIT_KELVIN:                        strcpy_s(propInfo.unit, unitNameSize, "KELVIN");                        break;
        case DCAMPROP_UNIT_METERPERSECOND:        strcpy_s(propInfo.unit, unitNameSize, "METERPERSECOND");        break;
        case DCAMPROP_UNIT_PERSECOND:                strcpy_s(propInfo.unit, unitNameSize, "PERSECOND");                break;
        case DCAMPROP_UNIT_DEGREE:                        strcpy_s(propInfo.unit, unitNameSize, "DEGREE");                        break;
```

```
        case DCAMPROP_UNIT_MICROMETER:                strcpy_s(propInfo.unit, unitNameSize, "MICROMETER");                break;
        default:                                                strcpy_s(propInfo.unit, unitNameSize, "NONE");                                                break;
        }

        return propInfo;
}

void fillCamProps(HDCAM hdcam)
{
        camProps.clear(); // Empty out the array
        int32 iProp = 0;        // property IDs
        DCAMPROPOPTION opt = DCAMPROP_OPTION_SUPPORT;
        while (!failed(dcamprop_getnextid(hdcam, &iProp, opt)))
        {
                camPropInfo propInfo = get_camPropInfo(hdcam, iProp);
                camProps.push_back(propInfo);
                if (propInfo.propAttr.attribute2 & DCAMPROP_ATTR2_ARRAYBASE)
                {
                        double nElem;
                        int32 iPropElemStep = propInfo.propAttr.iPropStep_Element;
                        dcamprop_getvalue(hdcam, propInfo.propAttr.iProp_NumberOfElement, &nElem);
                        for (int32 i = 1; i < int32(nElem); ++i)
                        {
                                camPropInfo propElemInfo = get_camPropInfo(hdcam, iProp + i * iPropElemStep);
                                camProps.push_back(propElemInfo);
                        }
                }
        }
}

void printCamPropsArray(std::ostream& out)
{
        out << std::setw(102) << std::setfill('-') << "" << std::endl << std::setfill(' ');
        out << "| " << std::right << std::setw(11) << "Property ID" << " | ";
        out << std::left << std::setw(propNameSize) << "Property Name" << " | ";
        out << std::right << std::setw(datatypeNameSize) << "Datatype" << " | ";
        out << std::right << std::setw(11) << "Value" << " | ";
        out << std::right << std::setw(unitNameSize) << "Unit" << " |" << std::endl;
        out << std::setw(102) << std::setfill('-') << "" << std::endl << std::setfill(' ');

        for (size_t i = 0; i < camProps.size(); ++i)
        {
                out << "| " << std::right << std::setw(11) << camProps[i].propID << " | ";
                out << std::left << std::setw(propNameSize) << camProps[i].propName << " | ";
                out << std::right << std::setw(datatypeNameSize) << camProps[i].dataType << " | ";
```

```
            if (trunc(camProps[i].currentVal) == camProps[i].currentVal)
                out << std::right << std::setw(11) << camProps[i].currentVal << " | ";
            else
                out << std::right << std::setw(11) << std::setprecision(4) << std::scientific << camProps[i].currentVal << std::defaultfloat << " | ";
            out << std::right << std::setw(unitNameSize) << camProps[i].unit << " |" << std::endl;
        }
        out << std::setw(102) << std::setfill('-') << "" << std::endl << std::setfill(' ');
}

void printCamPropInfo(std::ostream& out, size_t camPropsIdx)
{
        // Make sure that the idx is within the array size
        if (camPropsIdx < camProps.size())
        {
            out << "\tID: " << camProps[camPropsIdx].propID << std::endl; // Print out the ID of the property
            out << "\tName: " << camProps[camPropsIdx].propName << std::endl; // Print out the name of the property
            out << "\tAttributes: " << std::endl; // Print out the property attributes
            for (int i = 0; i < camProps[camPropsIdx].nAttr; ++i)
                out << "\t\t" << camProps[camPropsIdx].attrNames[i] << std::endl;
            out << "\tDatatype: " << camProps[camPropsIdx].dataType << std::endl; // Print out the property datatype
            if (camProps[camPropsIdx].nSuppVals >= 0) // List possible values for the property
            {
                out << "\tSupported Values: " << std::endl;
                for (int i = 0; i < camProps[camPropsIdx].nSuppVals; ++i)
                    out << "\t\t" << std::setw(8) << std::left << camProps[camPropsIdx].suppVals[i] << " " << camProps[camPropsIdx].suppValNames[i] << std::endl;
            }
            if (!std::isnan(camProps[camPropsIdx].min)) // Print out the minimum possible property values
                out << "\tMinimum: " << camProps[camPropsIdx].min << std::endl;
            if (!std::isnan(camProps[camPropsIdx].max)) // Print out the maximum possible property values
                out << "\tMaximum: " << camProps[camPropsIdx].max << std::endl;
            if (!std::isnan(camProps[camPropsIdx].step)) // Print out the step size of property values
                out << "\tStep: " << camProps[camPropsIdx].step << std::endl;
            if (!std::isnan(camProps[camPropsIdx].defaultVal)) // Print out the default property value
                out << "\tDefault Value: " << camProps[camPropsIdx].defaultVal << std::endl;
            if (!std::isnan(camProps[camPropsIdx].currentVal)) // Print out current property value
```

```
			out << "\tCurrent Value: " << camProps[camPropsIdx].currentVal << std::endl;
	}
	else
		out << "Invalid Camera Property Index. Update the camera property list to make sure that the property exists." << std::endl;
}

int getCamPropsIdxByName(std::string& propName)
{
	int propListIdx = -1;
	for (int i = 0; i < camProps.size(); ++i)
		if (propName.compare(camProps[i].propName) == 0)
		{
			propListIdx = i;
			break;
		}
	return propListIdx;
}

int getCamPropsIdxByID(int32 propID)
{
	int propListIdx = -1;
	for (int i = 0; i < camProps.size(); ++i)
		if (camProps[i].propID == propID)
		{
			propListIdx = i;
			break;
		}
	return propListIdx;
}

void setCamPropValue(HDCAM hdcam, std::ostream& out, size_t camPropsIdx, double val)
{
	DCAMERR err = DCAMERR_SUCCESS;
	double reqVal = val;
	do
	{
		val = reqVal;
		err = dcamprop_setgetvalue(hdcam, camProps[camPropsIdx].propID, &val);
		fillCamProps(hdcam);
	} while (err == DCAMERR_SUCCESS && camProps[camPropsIdx].currentVal != val);

	if (failed(err))
	{
		out << "Failed setting " << camProps[camPropsIdx].propName << " = " << reqVal << std::endl
			<< "Make sure that the value provided is valid." << std::endl;
		camProps[camPropsIdx] = get_camPropInfo(hdcam, camProps[camPropsIdx].propID);
		printCamPropInfo(out, camPropsIdx);
```

```
                return;
        }
}
```

H.2.6 camProp.h

```
#pragma once

#define NOMINMAX
#include "dcamMisc/console4.h"
#include <limits>
#include <iomanip>
#include <vector>

constexpr int propNameSize = 40;
constexpr int attrNameSize = 24;
constexpr int datatypeNameSize = 8;
constexpr int suppValNameSize = 24;
constexpr int unitNameSize = 16;

typedef struct _camPropInfo
{
        int32 propID = 0;
        DCAMPROP_ATTR propAttr;
        char propName[propNameSize];
        int nAttr = 0;
        char attrNames[16][attrNameSize];
        char dataType[datatypeNameSize];
        int nSuppVals = -1;
        int32 suppVals[32];
        char suppValNames[32][suppValNameSize];
        char unit[unitNameSize];
        double min = std::numeric_limits<double>::quiet_NaN();
        double max = std::numeric_limits<double>::quiet_NaN();
        double step = std::numeric_limits<double>::quiet_NaN();
        double defaultVal = std::numeric_limits<double>::quiet_NaN();
        double currentVal = std::numeric_limits<double>::quiet_NaN();
} camPropInfo;

extern std::vector<camPropInfo> camProps; // Array to store camera properties

camPropInfo get_camPropInfo(HDCAM hdcam, int32 propID);
void fillCamProps(HDCAM hdcam);
void printCamPropsArray(std::ostream& out);
void printCamPropInfo(std::ostream& out, size_t camPropsIdx);
int getCamPropsIdxByName(std::string& propName);
int getCamPropsIdxByID(int32 propID);
void setCamPropValue(HDCAM hdcam, std::ostream& out, size_t camPropsIdx, double val);
```

H.2.7 camRec.cpp

```
#include "camRec.h"

// Variable to indicate the current state of recording
std::atomic<bool> camRcrdng = false;

// Variable to indicate whether the camera is ready to start recording
std::atomic<bool> camRdy2Capt = false;

void recordFrames(std::ostream& out, HDCAM hdcam, HDCAMWAIT hwait)
{
    DCAMERR err;

    // start capture
    err = dcamcap_start(hdcam, DCAMCAP_START_SNAP);
    if (failed(err))
    {
        out << "Could not start camera recording." << std::endl;
        return;
    }
    else
    {
        // State that the camera is ready to capture frames
        camRdy2Capt = true;

        // set wait param
        DCAMWAIT_START waitstart;
        memset(&waitstart, 0, sizeof(waitstart));
        waitstart.size = sizeof(waitstart);
        waitstart.eventmask = DCAMWAIT_CAPEVENT_STOPPED | DCAMWAIT_CAPEVENT_RELOADFRAME;
        waitstart.timeout = 1000;

        // Wait for capture to complete
        bool bStop = false;
        while (!bStop)
        {
            err = dcamwait_start(hwait, &waitstart);
            if (!failed(err) && (waitstart.eventhappened & DCAMWAIT_CAPEVENT_STOPPED))
                bStop = true;
            if (!failed(err) && (waitstart.eventhappened & DCAMWAIT_CAPEVENT_RELOADFRAME))
                out << "DCAMWAIT_CAPEVENT_RELOADFRAME" << std::endl;

            // get capture and transfer status
            int32 capStatus = 0;
            err = dcamcap_status(hdcam, &capStatus);
            out << "Capture Status: ";
            if (failed(err))
                out << "Can't retrieve";
            else
                switch (capStatus)
```

```
                {
                case DCAMCAP_STATUS_BUSY: out << "BUSY"; break;
                case DCAMCAP_STATUS_ERROR: out << "ERROR"; break;
                case DCAMCAP_STATUS_READY: out << "READY"; break;
                case DCAMCAP_STATUS_STABLE: out << "STABLE"; break;
                case DCAMCAP_STATUS_UNSTABLE: out << "UNSTABLE"; break;
                default: break;
                }
            out << ", ";
            DCAMCAP_TRANSFERINFO transInfo;
            memset(&transInfo, 0, sizeof(transInfo));
            transInfo.size = sizeof(transInfo);
            transInfo.iKind = DCAMCAP_TRANSFERKIND_FRAME;
            err = dcamcap_transferinfo(hdcam, &transInfo);
            if (failed(err))
                out << "Frames Captured: Unavailable, Newest Frame Index: Unavailable" <<
                ↪ std::endl;
            else
                out << "Frames Captured: " << transInfo.nFrameCount << ", "
                << "Newest Frame Index: " << transInfo.nNewestFrameIndex << std::endl;
        }
        // stop capture
        dcamcap_stop(hdcam);

        // State that the camera is not ready to capture frames
        camRdy2Capt = false;
    }
}

void startCamRecording(std::ostream& out, HDCAM hdcam, camRecInfo recInfo)
{
    // State that this function is called
    camRcrdng = true;

    // Variable for string error codes
    DCAMERR err;

    // open wait handle
    DCAMWAIT_OPEN        waitopen;
    memset(&waitopen, 0, sizeof(waitopen));
    waitopen.size = sizeof(waitopen);
    waitopen.hdcam = hdcam;

    err = dcamwait_open(&waitopen);
    if (failed(err))
        out << "Could not create DCAMWAIT_OPEN object." << std::endl;
    else
    {
        HDCAMWAIT hwait = waitopen.hwait;

        // Get frame size in bytes
```

```
        double bufframebytes;
        err = dcamprop_getvalue(hdcam, DCAM_IDPROP_BUFFER_FRAMEBYTES, &bufframebytes);
        if (failed(err))
            out << "Could not retrieve DCAM_IDPROP_BUFFER_FRAMEBYTES value." << std::endl;
        else
        {
            // Check if enough memory is available
            MEMORYSTATUSEX mem;
            mem.dwLength = sizeof(mem);
            if (!GlobalMemoryStatusEx(&mem))
                out << "Failed to retrieve physical memory info." << std::endl;
            else
            {
                size_t frameSize = (size_t)bufframebytes;
                int        number_of_buffer = recInfo.nFrames;
                if (mem.ullAvailPhys < frameSize * number_of_buffer)
                    out << "Not enough memory. Required: " << frameSize * number_of_buffer
                    ↪  / 1024.0 / 1024.0 / 1024.0
                        << " GB, Available: " << mem.ullAvailPhys / 1024.0 / 1024.0 /
                        ↪  1024.0 << " GB" << std::endl;
                else
                {
                    // allocate buffer
                    void** pFrames = new void* [number_of_buffer];
                    char* buf = new char[frameSize * number_of_buffer];
                    memset(buf, 0, frameSize * number_of_buffer);

                    int                 i;
                    for (i = 0; i < number_of_buffer; i++)
                    {
                        pFrames[i] = buf + frameSize * i;
                    }

                    DCAMBUF_ATTACH bufattach;
                    memset(&bufattach, 0, sizeof(bufattach));
                    bufattach.size = sizeof(bufattach);
                    bufattach.iKind = DCAMBUF_ATTACHKIND_FRAME;
                    bufattach.buffer = pFrames;
                    bufattach.buffercount = number_of_buffer;

                    // attach user buffer
                    err = dcambuf_attach(hdcam, &bufattach);
                    if (failed(err))
                        out << "Could not attach frame buffer." << std::endl;
                    else
                    {
                        // Start recording
                        recordFrames(out, hdcam, hwait);

                        // release buffer
                        dcambuf_release(hdcam);
```

```cpp

                        // Save images
                        std::ofstream imgsFile(recInfo.filePath, std::ios::out | std::ios::binary);
                        if (!imgsFile)
                            out << "Could not open file: " << recInfo.filePath << std::endl;
                        else
                        {
                            double imgWidth_d = 0, imgHeight_d = 0;
                            DCAMERR errW = dcamprop_getvalue(hdcam, DCAM_IDPROP_IMAGE_WIDTH, &imgWidth_d);
                            DCAMERR errH = dcamprop_getvalue(hdcam, DCAM_IDPROP_IMAGE_HEIGHT, &imgHeight_d);
                            if (failed(errW) || failed(errH))
                                out << "Could not retrieve image width and height properties." << std::endl;
                            else
                            {
                                uint32_t imgWidth = (uint32_t)imgWidth_d, imgHeight = (uint32_t)imgHeight_d, nFrames = (uint32_t)recInfo.nFrames;
                                imgsFile.write((char*)&imgWidth, sizeof(imgWidth));
                                imgsFile.write((char*)&imgHeight, sizeof(imgHeight));
                                imgsFile.write((char*)&nFrames, sizeof(nFrames));
                                imgsFile.write(buf, frameSize * number_of_buffer);
                            }
                            imgsFile.close();
                            if (!imgsFile.good())
                                out << "Error occurred at writing time!" << std::endl;
                        }
                    }
                    // free buffer
                    delete[] buf;
                    delete[] pFrames;
                }
            }
        }
        // close wait handle
        dcamwait_close(hwait);
    }

    // State that this function call has ended
    camRcrdng = false;
}

void waitFinishCamRcrdng(std::ostream &out)
{
    if (camRcrdng)
        out << "Waiting for the camera to finish recording." << std::endl;
```

```
    while (camRcrdng);
}
```

H.2.8 camRec.h

```
#pragma once

#define NOMINMAX
#include "dcamMisc/console4.h"

#include <iostream>
#include <filesystem>
#include <fstream>

struct camRecInfo
{
        std::filesystem::path filePath;
        unsigned int nFrames = 0;
};

// Variable to indicate whether startCamRecording is currently executing
extern std::atomic<bool> camRcrdng;

// Variable to indicate whether the camera is ready to capture frames
extern std::atomic<bool> camRdy2Capt;

void startCamRecording(std::ostream& out, HDCAM hdcam, camRecInfo recInfo);
void waitFinishCamRcrdng(std::ostream &out);
```

H.2.9 camSoft.cpp

```
// camSoft.cpp : Defines the entry point for the application.
//

#include "camSoft.h"

HDCAM hdcam = NULL;

bool initCam()
{
        std::cout << "Looking for Hamamatsu C11440-22C camera." << std::endl;
        hdcam = dcamcon_init_open(); // Unitialize DCAM-API and open device
        if (hdcam != NULL)
                std::cout << "Camera initialization sucessful." << std::endl;
        else
                dcamapi_uninit(); // Uninitialize DCAM-API
        return (hdcam != NULL);
```

```
}

void otSoftCamRec(std::ostream& out, unsigned int nFrames, std::filesystem::path filePath)
{
	filePath.make_preferred();
	if (liveCapOn)
		out << "Live feed must be turned off before recording." << std::endl;
	else if (expsUnderProgress)
		out << "The ongoing experiments must be stopped before recording is started." << std::endl;
	else if (camRcrdng)
		out << "The camera is already recording something." << std::endl;
	else
	{
		if (filePath.extension() == ".bin")
		{
			camRecInfo recInfo;
			if (filePath.parent_path() == "")
			{
				recInfo.nFrames = nFrames;
				recInfo.filePath = filePath.filename();
				std::thread camRecThread(startCamRecording, std::ref(out), hdcam, recInfo);
				camRecThread.detach();
			}
			else if (std::filesystem::is_directory(filePath.parent_path()) || std::filesystem::create_directories(filePath.parent_path()))
			{
				recInfo.nFrames = nFrames;
				recInfo.filePath = filePath.parent_path() /= filePath.filename();
				std::thread camRecThread(startCamRecording, std::ref(out), hdcam, recInfo);
				camRecThread.detach();
			}
			else
				out << "Failed to create the path specified." << std::endl;
		}
		else
			out << "Recording file name must contain \".bin\" extension." << std::endl;
	}
}

void otSoftShtrCtrl(std::ostream& out, std::string cmd)
{
	sndShtrCmd(out, cmd);
}

void camPropsUpdate(std::ostream& out)
{
```

```
        fillCamProps(hdcam);
}

void camPropList(std::ostream& out)
{
        fillCamProps(hdcam);
        printCamPropsArray(out);
}

void camPropInfoByName(std::ostream& out, std::string propName)
{
        fillCamProps(hdcam);
        int propListIdx = getCamPropsIdxByName(propName);
        if (propListIdx >= 0)
                printCamPropInfo(out, propListIdx);
        else
                out << "Property with name \"" << propName << "\" doesn't exist." << std::endl;
}

void camPropInfoByID(std::ostream& out, int32 propID)
{
        fillCamProps(hdcam);
        int propListIdx = getCamPropsIdxByID(propID);
        if (propListIdx >= 0)
                printCamPropInfo(out, propListIdx);
        else
                out << "Property with ID \"" << propID << "\" doesn't exist." << std::endl;
}

void camPropSetByName(std::ostream& out, std::string propName, double val)
{
        if (liveCapOn || camRcrdng || expsUnderProgress)
                out << "Camera properties cannot be set while live feed is on, camera is recording or experiments are being conducted." << std::endl;
        else
        {
                fillCamProps(hdcam);
                int propListIdx = getCamPropsIdxByName(propName);
                if (propListIdx >= 0)
                        setCamPropValue(hdcam, out, propListIdx, val);
                else
                        out << "Property with name \"" << propName << "\" doesn't exist." << std::endl;
        }
}

void camPropSetByID(std::ostream& out, int32 propID, double val)
{
        if (liveCapOn || camRcrdng || expsUnderProgress)
```

```cpp
			out << "Camera properties cannot be set while live feed is on, camera is recording or experiments are being conducted." << std::endl;
		else
		{
			fillCamProps(hdcam);
			int propListIdx = getCamPropsIdxByID(propID);
			if (propListIdx >= 0)
				setCamPropValue(hdcam, out, propListIdx, val);
			else
				out << "Property with ID \"" << propID << "\" doesn't exist." << std::endl;
		}
}

void liveCapStart(std::ostream& out)
{
	if (liveCapOn)
		out << "Live feed from camera is already on." << std::endl;
	else if (expsUnderProgress)
		out << "Live feed cannot be started while experiments are being conducted." << std::endl;
	else if (camRcrdng)
		out << "Live feed cannot be started while the camera is recording." << std::endl;
	else
	{
		fillCamProps(hdcam);
		int propListWidthIdx = getCamPropsIdxByID(DCAM_IDPROP_IMAGE_WIDTH);
		int propListHeightIdx = getCamPropsIdxByID(DCAM_IDPROP_IMAGE_HEIGHT);
		setCamCapImgSize(camProps[propListWidthIdx].currentVal, camProps[propListHeightIdx].currentVal);
		std::thread liveCapThread(startCamCap, std::ref(out), hdcam);
		liveCapThread.detach();
	}
}

void liveCapStop(std::ostream& out)
{
	if (!liveCapOn)
		out << "Live feed from camera is already off." << std::endl;
	else
		stopCamCap(out);
}

void liveCapLUT(std::ostream& out, int lutMin, int lutMax)
{
	if (lutMax > 65535 || lutMax <= lutMin || lutMin < 0)
		out << "Make sure that lutMin >= 0, lutMax <= 65535 and lutMax > lutMin." << std::endl;
	else
		setCamCapLUT(lutMin, lutMax);
```

```
}

void condExpsStart(std::ostream &out, size_t numExps)
{
        if (liveCapOn)
                out << "Live feed must be turned off before conducting experiments." <<
                ↪  std::endl;
        else if (expsUnderProgress)
                out << "Experiments are already being conducted." << std::endl;
        else if (camRcrdng)
                out << "Experiments cannot be conducted while the camera is recording." <<
                ↪  std::endl;
        else
        {
                std::thread condExpsThread(startConductingExps, std::ref(out), numExps,
                ↪  hdcam);
                condExpsThread.detach();
        }
}

void condExpsStop(std::ostream &out)
{
        stopConductingExps(out);
}

void condExpsStatus(std::ostream &out)
{
        expsStatus(out);
}

int main(int argc, char* const argv[])
{
        // Roll the intro
        std::cout <<
        ↪  "////////////////////////////////////////////////////////////////////////////"
        ↪  << std::endl;
        std::cout << "// otSoft: A Command Line Program to perform optical tweezer
        ↪  experiments //" << std::endl;
        std::cout << "// Developed by: Vatsal Asitkumar Joshi
        ↪  //" << std::endl;
        std::cout << "// Hardware required: Hamamatsu C11440-22C Camera,
        ↪  //" << std::endl;
        std::cout << "//                    Raspberry Pi Pico Microcontroller,
        ↪  //" << std::endl;
        std::cout << "//                    1064nm Laser
        ↪  //" << std::endl;
        std::cout << "// Date last updated: 03/09/2022
        ↪  //" << std::endl;
        std::cout <<
        ↪  "////////////////////////////////////////////////////////////////////////////"
        ↪  << std::endl << std::endl;
```

```
        if (initShtr() && initCam())
        {
                fillCamProps(hdcam); // Get a list of camera properties
                std::cout << std::endl;

                // Create a root menu of our cli
                auto otSoft = std::make_unique<cli::Menu>("otSoft", "Main menu of this application.");
                otSoft->Insert("camRec", otSoftCamRec, "Start camera recording.");
                otSoft->Insert("shtrCtrl", otSoftShtrCtrl, "Passthrough for setting up shutter controller. Type 'shtrCtrl help' for more information.");

                // Create a submenu for camera properties
                auto camProp = std::make_unique<cli::Menu>("camProp", "Menu to access camera properties.");
                camProp->Insert("update", camPropsUpdate, "Update the list of all the properties of Hamamatsu C11440-22C camera.");
                camProp->Insert("list", camPropList, "List all the properties of Hamamatsu C11440-22C camera.");
                camProp->Insert("infoByName", camPropInfoByName, "Get info regarding a certain property by name.");
                camProp->Insert("infoByID", camPropInfoByID, "Get info regarding a certain property by ID.");
                camProp->Insert("setByName", camPropSetByName, "Set camera property value.");
                camProp->Insert("setByID", camPropSetByID, "Set camera property value.");
                otSoft->Insert(std::move(camProp));

                // Create a submenu for camera live feed
                auto liveCap = std::make_unique<cli::Menu>("liveCap", "Menu to show live feed from the camera.");
                liveCap->Insert("start", liveCapStart, "Start the camera live feed.");
                liveCap->Insert("stop", liveCapStop, "Stop the camera live feed.");
                liveCap->Insert("lut", liveCapLUT, "Update input-output mapping of the camera pixel values.");
                otSoft->Insert(std::move(liveCap));

                // Create a submenu for conducting experiments
                auto condExps = std::make_unique<cli::Menu>("condExps", "Menu for conducting experiments.");
                condExps->Insert("start", condExpsStart, "Start conducting n experiments.");
                condExps->Insert("stop", condExpsStop, "Stop ongoing experiments.");
                condExps->Insert("status", condExpsStatus, "Status of the ongoing experiments.");
                otSoft->Insert(std::move(condExps));

                // create the cli with the root menu
                cli::Cli cli(std::move(otSoft));

```

```
                // global exit action
                cli.ExitAction([](auto &out)
                                {
                                        stopConductingExps(out);        // Make
                                        ↪  sure no experiment is being
                                        ↪  conducted
                                        stopCamCap(out);
                                        ↪  // Make sure the camera is not
                                        ↪  capturing
                                        waitFinishCamRcrdng(out); // Wait for
                                        ↪  the camera to finish recording
                                        deinitShtr(out);
                                        ↪  // Reboot the shutter controller
                                        dcamdev_close(hdcam);        // close
                                        ↪  DCAM handle
                                        dcamapi_uninit();                //
                                        ↪  Uninit DCAM-API
                                        out << "Camera uninitialized properly."
                                        ↪  << std::endl;
                                        out << "Press Enter to exit...";
                                });

                cli::LoopScheduler scheduler;
                cli::CliLocalTerminalSession localSession(cli, scheduler, std::cout, 200);

                localSession.ExitAction(
                        [&scheduler](auto &out) // session exit action
                        {
                                scheduler.Stop();
                        });

                scheduler.Run();
        }
        else
                std::cout << "Shutter or Camera initialization unsucessful. Try again
                ↪  later." << std::endl;
}
```

H.2.10 camSoft.h

```
// camSoft.h : Include file for standard system include files,
// or project specific include files.

#pragma once

#include <iostream>

// TODO: Reference additional headers your program requires here.
#include "dcamMisc/common.h"
```

```
#include "camProp.h"
#include "liveCap.h"
#include "camRec.h"
#include "shtrCtrl.h"
#include "condExps.h"
#include "cli/include/cli/cli.h"
#include "cli/include/cli/loopscheduler.h"
#include "cli/include/cli/clilocalsession.h"
#include <iomanip>
#include <thread>
#include <filesystem>
```

H.2.11 CMakeLists.txt

```
# CMakeList.txt : CMake project for camSoft, include source and define
# project specific logic here.
#
cmake_minimum_required (VERSION 3.15)

project ("camSoft")

# Make sure that dll runtime libraries are used
set(CMAKE_MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>")

# Add source to this project's executable.
add_executable (camSoft
"dcamsdk4/inc/dcamapi4.h" "dcamsdk4/inc/dcamprop.h"
"dcamMisc/common.cpp" "dcamMisc/common.h" "dcamMisc/console4.h"
"serialib/lib/serialib.cpp" "serialib/lib/serialib.h"

"camProp.cpp" "camProp.h"
"camRec.cpp" "camRec.h"
"liveCap.cpp" "liveCap.h"
"shtrCtrl.cpp" "shtrCtrl.h"
"condExps.cpp" "condExps.h"
"camSoft.cpp" "camSoft.h"
)

if (CMAKE_VERSION VERSION_GREATER 3.15)
  set_property(TARGET camSoft PROPERTY CXX_STANDARD 17)
endif()

# Define locations to the header files of different libraries
target_include_directories(camSoft PUBLIC
"${CMAKE_SOURCE_DIR}/glm"
"${CMAKE_SOURCE_DIR}/glew-2.1.0/include"
"${CMAKE_SOURCE_DIR}/glfw-3.3.8.bin.WIN64/include"
)

```

```
# Look for DCAMAPI library
find_library(
        DCAMAPI_LIB REQUIRED
        NAMES dcamapi # Name of the file to look for
        PATHS "${CMAKE_SOURCE_DIR}/dcamsdk4/lib/win64" # Folder to look into
        NO_DEFAULT_PATH # Do not search system default paths
)
message(STATUS "DCAMAPI_LIB: [${DCAMAPI_LIB}]")

# Look for OPENGL library
find_package(OpenGL REQUIRED)

# Look for GLEW library
find_library(
        GLEW_LIB REQUIRED
        NAMES glew32s # Name of the file to look for
        PATHS "${CMAKE_SOURCE_DIR}/glew-2.1.0/lib/Release/x64" # Folder to look into
        NO_DEFAULT_PATH # Do not search system default paths
)
message(STATUS "GLEW_LIB: [${GLEW_LIB}]")

# Look for GLFW library
find_library(
        GLFW_LIB REQUIRED
        NAMES glfw3_mt # Name of the file to look for
        PATHS "${CMAKE_SOURCE_DIR}/glfw-3.3.8.bin.WIN64/lib-vc2022" # Folder to look into
        NO_DEFAULT_PATH # Do not search system default paths
)
message(STATUS "GLFW_LIB: [${GLFW_LIB}]")

# Link all the libraries
target_link_libraries(camSoft PUBLIC
${DCAMAPI_LIB}
${GLEW_LIB}
${GLFW_LIB}
OpenGL::GL
)
```

H.2.12 condExps.cpp

```
#include "condExps.h"

// Define globals for this file only
size_t nExps = 0; // Number of experiments to be conducted
size_t expLen_us = 0; // Length of one experiment in us

// Define globals that are 'extern' in the header file
std::atomic<bool> expsUnderProgress = false; // Whether the experiments are currently being
↪  conducted or not
```

```cpp

void startConductingExps(std::ostream &out, size_t numExps, HDCAM hdcam)
{
    // State that some experiments are being conducted
    expsUnderProgress = true;

    // Get some info from the shutter controller
    size_t expLen_us = std::stoull(sndCmdRecRspShtr(std::string("++expLen"))); // Length of one experiment in us
    size_t camTrigLen_us = std::stoull(sndCmdRecRspShtr(std::string("++camTrigLen"))); // Length of time the camera should be recording in us

    // Define the number of experiments
    nExps = numExps;

    // Get some camera info
    size_t exposureTime_ns = (size_t)(get_camPropInfo(hdcam, DCAM_IDPROP_EXPOSURETIME).currentVal * 1e9);
    size_t nFrames = camTrigLen_us * 1000 / exposureTime_ns;

    // Get camera property values so that we can reset it after conducting experiments
    double trigSource = get_camPropInfo(hdcam, DCAM_IDPROP_TRIGGERSOURCE).currentVal;
    double trigMode = get_camPropInfo(hdcam, DCAM_IDPROP_TRIGGER_MODE).currentVal;
    double trigPolarity = get_camPropInfo(hdcam, DCAM_IDPROP_TRIGGERPOLARITY).currentVal;

    // Name a folder with current date and time
    char fldrName[80];
    time_t t = std::time(nullptr);
    struct tm timeInfo;
    localtime_s(&timeInfo, &t);
    std::strftime(fldrName, sizeof(fldrName), "%Y%m%d%H%M%S", &timeInfo);
    std::filesystem::path folderPath = std::string(fldrName);

    // If the folder exists then delete it, probably won't ever happen.
    if (std::filesystem::is_directory(folderPath))
        std::filesystem::remove_all(folderPath);

    // Create a folder
    std::filesystem::create_directories(folderPath);

    // Start conducting experiments
    while (nExps && expsUnderProgress)
    {
        // Set up the recording stuff
        std::filesystem::path filePath = (folderPath / (std::to_string(numExps - nExps) + std::string(".bin"))).make_preferred();
        if (std::filesystem::is_directory(filePath.parent_path())) // Make sure if the folder was created
        {
            camRecInfo recInfo;
            recInfo.nFrames = (unsigned int) nFrames;
```

```
            recInfo.filePath = filePath.parent_path() /= filePath.filename();
            // Set camera trigger source to external
            setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGERSOURCE),
            ↪  DCAMPROP_TRIGGERSOURCE__EXTERNAL);
            setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGER_MODE),
            ↪  DCAMPROP_TRIGGER_MODE__START);
            setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGERPOLARITY),
            ↪  DCAMPROP_TRIGGERPOLARITY__POSITIVE);
            // Initiate the camera capture
            std::thread camRecThread(startCamRecording, std::ref(out), hdcam, recInfo);
            while (!camRcrdng); // Wait for the thread to actually start executing
            camRecThread.detach();
        }
        else
            out << "Failed to create the folder." << std::endl;

        // Wait till camera is ready to capture or an error ocurred
        while (camRcrdng && !camRdy2Capt);

        // Make the shutter controller execute one experiment
        if (camRcrdng)
            sndShtrCmd(out, std::string("++start"));
        else
            out << "Seems like the camera could not be set up properly for Exp No.: " <<
            ↪  nExps << std::endl;

        // If an experiment is started then code should not reach here before it is
        ↪  completed.
        // Wait for the camera to finish recording, transferring images to SSD/HDD should
        ↪  be remaining at this point.
        waitFinishCamRcrdng(out);

        // Reduce nExps by one only if it is grather than 0
        nExps += nExps ? -1 : 0;
    }

    // Reset Camera properties to the original values
    setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGERSOURCE), trigSource);
    setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGER_MODE), trigMode);
    setCamPropValue(hdcam, out, getCamPropsIdxByID(DCAM_IDPROP_TRIGGERPOLARITY),
    ↪  trigPolarity);

    // State that the experiments are done
    expsUnderProgress = false;

    // Make sure that the number of experiments is zero
    nExps = 0;
}

void stopConductingExps(std::ostream &out)
{
```

```
    nExps = 0;
    if (expsUnderProgress)
        out << "The system will stop after finishing the ongoing experiment." << std::endl;
    while (expsUnderProgress);
}

void expsStatus(std::ostream &out)
{
    if (expsUnderProgress)
    {
        out << "Number of experements remaining to be conducted:    " << nExps << std::endl;
        out << "Time required to finish remaining experiments:      " << expLen_us * nExps / 1000 << " ms at least" << std::endl;
    }
    else
        out << "No experiments are under process." << std::endl;
}
```

H.2.13 condExps.h

```
#pragma once

#define NOMINMAX
#include <iostream>
#include <ctime>
#include <thread>
#include "camProp.h"
#include "shtrCtrl.h"
#include "camRec.h"

extern std::atomic<bool> expsUnderProgress; // Whether the experiments are currently being conducted or not

void startConductingExps(std::ostream &out, size_t numExps, HDCAM hdcam);
void stopConductingExps(std::ostream &out);
void expsStatus(std::ostream &out);
```

H.2.14 liveCap.cpp

```
#include "liveCap.h"

// Define globals for this file only
GLsizei liveCapImgWidth = 2048, liveCapImgHeight = 2048; // Global variables to store image size
double liveCapOffsetX = 0, liveCapOffsetY = 0, liveCapScale = 1, liveCapWinPxPerImPx = 1; // Global variables for pan and zoom
```

```cpp
GLfloat liveCapLutMin = 0, liveCapLutMax = 65535; // Global variables to define LUT
GLint liveCapLutMinUniformLoc = -1, liveCapLutMaxUniformLoc = -1; // Global uniform location for LUT values
GLint liveCapMVPUniformLoc = -1; // Global uniform location for MVP matrix
bool liveFeedStopped = true; // Variable stating whether the live feed is stopped or not

// Define globals that are 'extern' in the header file
std::atomic<bool> liveCapOn = false; // Global variable to indicate live capture state

GLuint CompileShader(GLuint type, const std::string& source)
{
    GLuint id = glCreateShader(type);
    const char* src = source.c_str();
    GLCall(glShaderSource(id, 1, &src, nullptr));
    GLCall(glCompileShader(id));

    int result;
    GLCall(glGetShaderiv(id, GL_COMPILE_STATUS, &result));
    if (result == GL_FALSE)
    {
        int length;
        GLCall(glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length));
        char* message = (char*)_malloca(length * sizeof(char));
        GLCall(glGetShaderInfoLog(id, length, &length, message));
        std::cout << "Failed to compile " << (type == GL_VERTEX_SHADER ? "vertex" : "fragment") << " shader: " << message << std::endl;
        GLCall(glDeleteShader(id));
        return 0;
    }

    return id;
}

GLuint CreateShader()
{
    std::string vertexShader =
        "#version 330 core\n"
        "\n"
        "layout(location = 0) in vec2 position;\n"
        "layout(location = 1) in vec2 texCoord;\n"
        "uniform mat4 u_MVP;\n"
        "\n"
        "out vec2 v_TexCoord;\n"
        "void main()\n"
        "{\n"
        "   gl_Position = u_MVP * vec4(position,0.0,1.0);\n"
        "   v_TexCoord = texCoord;\n"
        "}\n";
```

```
    std::string fragmentShader =
        "#version 330 core\n"
        "\n"
        "out vec3 FragColor;\n"
        "in vec2 v_TexCoord;\n"
        "uniform sampler2D u_imgTex;\n"
        "uniform float u_lutMin;\n"
        "uniform float u_lutMax;\n"
        "void main()\n"
        "{\n"
        "   float colorVal = texture(u_imgTex,v_TexCoord).r;\n"
        //"   colorVal *= 600;\n"
        "   colorVal = (65535*colorVal - u_lutMin) / (u_lutMax - u_lutMin);\n"
        "   colorVal = colorVal > 1 ? 1 : colorVal < 0 ? 0 : colorVal;\n"
        "   FragColor = vec3(colorVal);\n"
        "}\n";

    GLuint program = glCreateProgram();
    GLuint vs = CompileShader(GL_VERTEX_SHADER, vertexShader);
    GLuint fs = CompileShader(GL_FRAGMENT_SHADER, fragmentShader);

    GLCall(glAttachShader(program, vs));
    GLCall(glAttachShader(program, fs));
    GLCall(glLinkProgram(program));
    GLCall(glValidateProgram(program));

    GLCall(glDeleteShader(vs));
    GLCall(glDeleteShader(fs));

    return program;
}

void applyMVP(float width, float height)
{
    glm::mat4 mvp = glm::ortho(-width / 2, width / 2, height / 2, -height / 2); // Create orthogonal Projection matrix
    glm::mat4 model = glm::mat4(1.0);
    model = glm::scale(model, glm::vec3(liveCapScale * liveCapWinPxPerImPx, liveCapScale * liveCapWinPxPerImPx, 1.0f));
    model = glm::translate(model, glm::vec3(liveCapOffsetX, liveCapOffsetY, 0.0f));
    mvp = mvp * model;
    if (liveCapMVPUniformLoc >= 0)
        GLCall(glUniformMatrix4fv(liveCapMVPUniformLoc, 1, GL_FALSE, &mvp[0][0]));
}

void applyLUT()
{
    static float lutMinLast = 0, lutMaxLast = 0;
    if ((lutMinLast != liveCapLutMin || lutMaxLast != liveCapLutMax) && liveCapLutMinUniformLoc >= 0 && liveCapLutMaxUniformLoc >= 0)
```

```
    {
        GLCall(glUniform1f(liveCapLutMinUniformLoc, liveCapLutMin));
        GLCall(glUniform1f(liveCapLutMaxUniformLoc, liveCapLutMax));
        lutMinLast = liveCapLutMin; lutMaxLast = liveCapLutMax;
    }
}

void handleWindowResize(GLFWwindow* window, int width, int height)
{
    double liveCapImgAspRatio = liveCapImgWidth / (float)liveCapImgHeight;
    double liveCapWinAspRatio = width / (float)height;
    if (liveCapImgAspRatio > liveCapWinAspRatio)
        liveCapWinPxPerImPx = width / (float)liveCapImgWidth;
    else
        liveCapWinPxPerImPx = height / (float)liveCapImgHeight;
    liveCapOffsetX = 0; liveCapOffsetY = 0; liveCapScale = 1;

    applyMVP((float)width, (float)height);

    GLCall(glViewport(0, 0, width, height));
}

void handleCursorPosition(GLFWwindow* window, double xpos, double ypos)
{
    static bool buttonPressed = false;
    static double dragX = 0, dragY = 0;

    int width, height;
    glfwGetWindowSize(window, &width, &height);
    //ypos = height - ypos;

    if (!buttonPressed && glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_PRESS)
    {
        dragX = xpos; dragY = ypos;
        buttonPressed = true;
    }
    else if (buttonPressed && glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) ==
    ↪  GLFW_PRESS)
    {
        liveCapOffsetX += (xpos - dragX) / liveCapScale / liveCapWinPxPerImPx;
        liveCapOffsetY += (ypos - dragY) / liveCapScale / liveCapWinPxPerImPx;
        applyMVP((float)width, (float)height);
        dragX = xpos; dragY = ypos;
    }
    else if (glfwGetMouseButton(window, GLFW_MOUSE_BUTTON_LEFT) == GLFW_RELEASE)
    {
        dragX = 0; dragY = 0;
        buttonPressed = false;
    }
}

```

```
void handleMouseScroll(GLFWwindow* window, double xoffset, double yoffset)
{
    double xpos, ypos;
    glfwGetCursorPos(window, &xpos, &ypos);

    int width, height;
    glfwGetWindowSize(window, &width, &height);
    //ypos = height - ypos;

    glm::mat4 projection = glm::ortho(-(float)width / 2, (float)width / 2, -(float)height / 2, (float)height / 2); // Create orthogonal Projection matrix
    glm::vec3 imgPx = glm::unProject(glm::vec3(xpos, ypos, 0.0f), glm::mat4(1.0), projection, glm::vec4(0, 0, width, height));

    liveCapOffsetX += -imgPx.x / liveCapScale / liveCapWinPxPerImPx;
    liveCapOffsetY += -imgPx.y / liveCapScale / liveCapWinPxPerImPx;
    liveCapScale += 0.1 * liveCapScale * yoffset;
    liveCapScale = liveCapScale < 0.1 ? 0.1 : liveCapScale;
    liveCapScale = liveCapScale > 100 ? 100 : liveCapScale;
    liveCapOffsetX += imgPx.x / liveCapScale / liveCapWinPxPerImPx;
    liveCapOffsetY += imgPx.y / liveCapScale / liveCapWinPxPerImPx;

    applyMVP((float)width, (float)height);
}

void handleWindowClose(GLFWwindow* window)
{
    glfwSetWindowShouldClose(window, GLFW_FALSE);
}

void handleKeyPress(GLFWwindow* window, int key, int scancode, int action, int mods)
{
    if (key == GLFW_KEY_SPACE)
    {
        int width, height;
        glfwGetWindowSize(window, &width, &height);
        liveCapOffsetX = 0; liveCapOffsetY = 0; liveCapScale = 1;
        applyMVP((float)width, (float)height);
        GLCall(glViewport(0, 0, width, height));
    }

}

void liveCapShow(std::ostream& out, HDCAM hdcam)
{
    DCAMERR camErr;

    // prepare frame param
    DCAMBUF_FRAME bufframe;
    memset(&bufframe, 0, sizeof(bufframe));
    bufframe.size = sizeof(bufframe);
```

```
bufframe.iFrame = 0;

GLFWwindow* window;

/* Initialize the library */
if (!glfwInit())
{
    out << "GLFW initialization failed." << std::endl << "otSoft> ";
    return;
}

glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

// Get Monitor information to set correct window size
GLFWmonitor* primaryMonitor = glfwGetPrimaryMonitor();
const GLFWvidmode* mode = glfwGetVideoMode(primaryMonitor);
int liveCapWindowWidth = (int)(mode->width / 1.5);
int liveCapWindowHeight = (int)(mode->height / 1.5);

/* Create a windowed mode window and its OpenGL context */
window = glfwCreateWindow(liveCapWindowWidth, liveCapWindowHeight, "liveCam", NULL, NULL);
if (!window)
{
    out << "GLFW window creation failed." << std::endl;
    glfwTerminate();
    return;
}

// Set necessary callbacks
glfwSetWindowSizeCallback(window, handleWindowResize);
glfwSetCursorPosCallback(window, handleCursorPosition);
glfwSetScrollCallback(window, handleMouseScroll);
glfwSetWindowCloseCallback(window, handleWindowClose);
glfwSetKeyCallback(window, handleKeyPress);

/* Make the window's context current */
glfwMakeContextCurrent(window);
glfwSwapInterval(1);


GLenum err = glewInit();
if (GLEW_OK != err)
{
    out << "GLEW initialization failed." << std::endl;
    out << "Error: " << glewGetErrorString(err) << std::endl;
    glfwTerminate();
    return;
}
```

```
    // Dark blue background
    glClearColor(0.251f, 0.259f, 0.345f, 1.0f);

    float vertices[16] = {
       -(liveCapImgWidth / 2.0f), -(liveCapImgHeight / 2.0f), 0.0f, 0.0f,
        (liveCapImgWidth / 2.0f), -(liveCapImgHeight / 2.0f), 1.0f, 0.0f,
        (liveCapImgWidth / 2.0f),  (liveCapImgHeight / 2.0f), 1.0f, 1.0f,
       -(liveCapImgWidth / 2.0f),  (liveCapImgHeight / 2.0f), 0.0f, 1.0f
    };

    GLuint indices[16] = {
       0, 1, 2,
       2, 3, 0
    };

    GLuint vao;
    GLCall(glGenVertexArrays(1, &vao));
    GLCall(glBindVertexArray(vao));

    GLuint vbo;
    GLCall(glGenBuffers(1, &vbo));
    GLCall(glBindBuffer(GL_ARRAY_BUFFER, vbo));
    GLCall(glBufferData(GL_ARRAY_BUFFER, sizeof(float) * 16, vertices, GL_STATIC_DRAW));

    GLCall(glEnableVertexAttribArray(0));
    GLCall(glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), 0));
    GLCall(glEnableVertexAttribArray(1));
    GLCall(glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(2 *
    ↪  sizeof(float))));

    GLuint ibo;
    GLCall(glGenBuffers(1, &ibo));
    GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo));
    GLCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint) * 6, indices,
    ↪  GL_STATIC_DRAW));

    GLuint shader = CreateShader();
    GLCall(glUseProgram(shader));

    /* OpenGL texture binding of the image from the camera  */
    GLuint imgTexID;
    GLCall(glGenTextures(1, &imgTexID)); /* Texture name generation */
    GLCall(glActiveTexture(GL_TEXTURE0)); // Activate texture unit 0 to store the image
    GLCall(glBindTexture(GL_TEXTURE_2D, imgTexID)); /* Binding of texture name */
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)); /* We will
    ↪  use linear interpolation for magnification filter */
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)); /* We will
    ↪  use linear interpolation for minifying filter */
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
```

```
    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_R16, liveCapImgWidth, liveCapImgHeight, 0,
    ↪  GL_RED, GL_UNSIGNED_SHORT, bufframe.buf));

    // Get uniform location to update MVP matrix
    liveCapMVPUniformLoc = glGetUniformLocation(shader, "u_MVP");

    // Get uniform location to update lutMin and lutMax
    liveCapLutMinUniformLoc = glGetUniformLocation(shader, "u_lutMin");
    liveCapLutMaxUniformLoc = glGetUniformLocation(shader, "u_lutMax");
    GLCall(glUniform1f(liveCapLutMinUniformLoc, liveCapLutMin));
    GLCall(glUniform1f(liveCapLutMaxUniformLoc, liveCapLutMax));

    // Call window resize function to correctly initialize MVP
    handleWindowResize(window, liveCapWindowWidth, liveCapWindowHeight);

    // State that the live feed is started
    liveFeedStopped = false;

    /* Loop until the user closes the window */
    while (!liveFeedStopped)
    {
        /* Render here */
        GLCall(glClear(GL_COLOR_BUFFER_BIT));

        // Update the texture
        // access image
        camErr = dcambuf_lockframe(hdcam, &bufframe);
        if (!failed(camErr) && bufframe.type == DCAM_PIXELTYPE_MONO16)
            GLCall(glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, liveCapImgWidth,
            ↪  liveCapImgHeight, GL_RED, GL_UNSIGNED_SHORT, bufframe.buf));
        else
            out << std::endl << "Error locking the camera frame." << std::endl;

        // Update LUT
        applyLUT();

        // Draw the triangles
        GLCall(glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, nullptr));

        /* Swap front and back buffers */
        glfwSwapBuffers(window);

        /* Poll for and process events */
        glfwPollEvents();
    }

    // Cleanup VBO
    GLCall(glDeleteTextures(1, &imgTexID));
    GLCall(glDeleteBuffers(1, &vbo));
    GLCall(glDeleteBuffers(1, &ibo));
    GLCall(glDeleteVertexArrays(1, &vao));
```

```
    GLCall(glDeleteProgram(shader));

    glfwTerminate();
    return;
}

void setCamCapImgSize(double width, double height)
{
    liveCapImgWidth = (GLsizei)width;
    liveCapImgHeight = (GLsizei)height;
}

void setCamCapLUT(int min, int max)
{
    liveCapLutMin = (GLfloat)min;
    liveCapLutMax = (GLfloat)max;
}

void startCamCap(std::ostream& out, HDCAM hdcam)
{
    // State that live capture has started
    liveCapOn = true;

    // Variable to hold camera errors
    DCAMERR err;

    // open wait handle
    DCAMWAIT_OPEN        waitopen;
    memset(&waitopen, 0, sizeof(waitopen));
    waitopen.size = sizeof(waitopen);
    waitopen.hdcam = hdcam;

    err = dcamwait_open(&waitopen);
    if (failed(err))
    {
        out << "Could not create DCAMWAIT_OPEN object." << std::endl;
        return;
    }
    else
    {
        HDCAMWAIT hwait = waitopen.hwait;

        // allocate buffer
        int32 number_of_buffer = 1;
        err = dcambuf_alloc(hdcam, number_of_buffer);
        if (failed(err))
        {
            out << "Could not allocate frame buffer in the DCAM module." << std::endl;
            return;
        }
        else
```

```
        {
            // start capture
            err = dcamcap_start(hdcam, DCAMCAP_START_SEQUENCE);
            if (failed(err))
            {
                out << "Could not start camera capture." << std::endl;
                return;
            }
            else
            {
                // set wait param
                DCAMWAIT_START waitstart;
                memset(&waitstart, 0, sizeof(waitstart));
                waitstart.size = sizeof(waitstart);
                waitstart.eventmask = DCAMWAIT_CAPEVENT_FRAMEREADY;
                waitstart.timeout = 11000;

                // wait image
                err = dcamwait_start(hwait, &waitstart);
                if (failed(err))
                {
                    out << "Could not detect frame ready event." << std::endl;
                    return;
                }

                // Start the live feed
                liveCapShow(out, hdcam);

                // stop capture
                dcamcap_stop(hdcam);
            }

            // release buffer
            dcambuf_release(hdcam);
        }

        // close wait handle
        dcamwait_close(hwait);
    }

    // State that the liveCap is stopped
    liveCapOn = false;
}

void stopCamCap(std::ostream &out)
{
    if (liveCapOn)
    {
        out << "Stopping the live feed." << std::endl;
        liveFeedStopped = true;
        setCamCapLUT();
```

```
    }

    while (liveCapOn);
}
```

H.2.15 liveCap.h

```
#pragma once

#define NOMINMAX
#include "dcamMisc/console4.h"

#define GLEW_STATIC
#include <GL/glew.h>

#include <GLFW/glfw3.h>
#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"

#include <iostream>
#include <vector>

// Macro to print opengl errors
#define GLCall(x)                               \
do {                                            \
    while(glGetError());                        \
    x;                                          \
    while(GLenum err = glGetError())            \
    {                                           \
        std::cout << "[OpenGL Error] "          \
                  << err << ": "                \
                  << #x << " in "               \
                  << __FILE__ << " at "         \
                  << __LINE__                   \
                  << std::endl;                 \
        __debugbreak();                         \
    }                                           \
} while (0)

// Global variable to indicate live capture state
extern std::atomic<bool> liveCapOn;

void setCamCapImgSize(double width, double height);
void setCamCapLUT(int min = 0, int max = 65535);
void startCamCap(std::ostream &out, HDCAM hdcam);
void stopCamCap(std::ostream &out);
```

H.2.16 shtrCtrl.cpp

```
#include "shtrCtrl.h"

serialib shtrPort; // Serial connection to the uC that controls the shutter and camera trigger

// Blocking function, if timeput_ms = 0, that returns with a character when available.
// Otherwise the number would be negative if no character is received within timeout.
int getcShtr(unsigned int timeout_ms)
{
    char c;
    int retCode = 0;
    if (timeout_ms)
        retCode = shtrPort.readChar(&c,timeout_ms);
    else
        while ((retCode = shtrPort.readChar(&c, 0)) < 1);
    if (retCode == 1)
        return c;
    else
        return -2;
}

// Blocking function, if timeput_ms = 0, that reads a string when available.
// The timeout_ms value is used for each character.
bool getsShtr(char *s, unsigned int timeout_ms)
{
    uint8_t count = 0;
    int c;

    while (count < MAX_CHARS)
    {
        c = getcShtr(timeout_ms);
        if (c == 8 || c == 127)
        {
            if (count == 0)
                continue;
            count--;
        }
        else if (c == 10 || c == 13) // Putty generally sends '\r' on Enter key
        {
            s[count] = '\0'; // Add Null character
            c = getcShtr(1000);            // Flush out the expected '\n' character if received within 1ms
            break;
        }
        else if (c > 31 && c < 127)
            s[count++] = c;
        else if (!timeout_ms)
            continue;
        else
```

```
            return 0;
    }
    return 1;
}

void flshShtrCommBuffer()
{
    shtrPort.flushReceiver();
}

bool initShtr()
{
    // Look for the ports
    std::string devName;
    bool correctToken = false;
    char token[MAX_CHARS];

    for (int i = 1; i < 99; i++)
    {
        // Prepare the port name (Windows)
        devName = "\\\\.\\COM" + std::to_string(i);

        // try to connect to the device
        if (shtrPort.openDevice(devName.c_str(), 115200) == 1)
        {
            // Set DTR and unset RTS
            shtrPort.DTR(true);
            shtrPort.RTS(false);

            // Read string from the port
            shtrPort.flushReceiver();
            for (size_t j = 0; j < 5; ++j)
            {
                int nBytes = getsShtr(token,200);
                correctToken = !strcmp(token, recToken);
                if (correctToken)
                    break;
            }

            if (correctToken)
            {
                shtrPort.writeString(sndToken);
                break;
            }
            else
                shtrPort.closeDevice();
        }
    }
    if (correctToken)
        std::cout << "Shutter controller detected on " << devName << "." << std::endl;
    else
```

```
        std::cout << "No shutter controller was detected." << std::endl;

    return correctToken;
}

void sndShtrCmd(std::ostream &out, std::string& s)
{
    // Flush the input buffer
    shtrPort.flushReceiver();

    // Append CR/LF combo to the string
    s = s + "\r\n";

    // Send the command
    putsShtr(s.c_str());

    // Receive the response
    char response[MAX_CHARS];
    while (getsShtr(response))
    {
        if (!strcmp(response,endRspToken))
            break;
        else
            out << std::string(response) << std::endl;
    }
}

std::string sndCmdRecRspShtr(std::string& s)
{
    // Flush the input buffer
    shtrPort.flushReceiver();

    // Append CR/LF combo to the string
    s = s + "\r\n";

    // Send the command
    putsShtr(s.c_str());

    // Receive the response and append it to a string
    char response[MAX_CHARS];
    std::string rsp;
    while (getsShtr(response))
    {
        if (!strcmp(response, endRspToken))
            break;
        else
            rsp += response;
    }

    return rsp;
}
```

```
void deinitShtr(std::ostream &out)
{
    sndShtrCmd(out, std::string("++reboot"));
    shtrPort.closeDevice();
}
```

H.2.17 shtrCtrl.h

```
#pragma once

#define NOMINMAX
#include <iostream>
#include <string>
#include "serialib/lib/serialib.h"

#define MAX_CHARS 80

// Define tokens used for communication
#define sndToken "116119101101122101114083111102116119097114101\r\n"    // Each 3 digits
↪   converted to chars results in 'tweezerSoftware'
#define recToken "115104117116116101114067111110116114111108108101114"  // Each 3 digits
↪   converted to chars results in 'shutterController'
#define endRspToken "101110100067109100082115112"                       // Each 3 digits
↪   converted to chars results in 'endCmdRsp'

extern serialib shtrPort; // Serial connection to the uC that controls the shutter and
↪   camera trigger

#define putcShtr(c) shtrPort.writeChar(c)   // Blocking function that writes a character
#define putsShtr(s) shtrPort.writeString(s) // Blocking function that writes a string

void flshShtrCommBuffer();
int getcShtr(unsigned int timeout_ms = 0);
bool getsShtr(char *s, unsigned int timeout_ms = 0);

bool initShtr();
void sndShtrCmd(std::ostream &out, std::string& s);
std::string sndCmdRecRspShtr(std::string &s);
void deinitShtr(std::ostream &out);
```

APPENDIX I

Code Listings for the Image/Video Handling

I.1 Directory tree

- `/` .... Add this folder to MATLAB path for easy access
  - `bin2avi.m` .... Compress raw data from the camera into an AVI video
  - `bin2jp2.m` .... Compress raw data from the camera into a SuperFrame image
  - `bin2mj2.m` .... Compress raw data from the camera into an MJ2 video
  - `jp2ToAvi.m` .... Create playable video from the SuperFrame image
  - `mj2ToAvi.m` .... Create playable video from the MJ2 video
  - `readImgsBin.m` .... Load images from the raw camera data
  - `readImgsJp2.m` .... Load images from the SuperFrame image
  - `readImgsMj2.m` .... Load images from the MJ2 video

I.2 Code Listings

I.2.1 `bin2avi.m`

```
function bin2avi(binFilePath,slowness,videoFilePath)
% This function assumes that the camera is capturing in the center of the
% sensor. If this is not the case then the calculation of the exposure time
% will be wrong.
arguments
    binFilePath {mustBeFile}
    slowness (1,1) double = 1
    videoFilePath (1,1) string = binFilePath
end

% Read jp2 files
imgs = readImgsBin(binFilePath);

% Adjust the contrast
imgs = imadjustn(imgs,stretchlim(imgs(:),0.01));

% Create an output file
[fPath,fName,~] = fileparts(videoFilePath);
vOut = VideoWriter(fullfile(fPath,fName),"Grayscale AVI");
vOut.FrameRate = 60;
open(vOut);

expTime = size(imgs,1)/2*9.74e-6; % Taken from camera datasheet
for i = 1:max(ceil(1/expTime/60/slowness),1):size(imgs,3)
    writeVideo(vOut,im2uint8(imgs(:,:,i)));
end

close(vOut);

end
```

I.2.2 bin2jp2.m

```
function bin2jp2(filePath)
arguments
    filePath {mustBeFile}
end

% Load the images from binary file
imgs = readImgsBin(filePath);

if ~isempty(imgs)
    % Create a folder with the name of the file
    [fPath,fName,~] = fileparts(filePath);
    folderPath = fullfile(fPath,fName);
    if ~exist(folderPath, "dir")
        mkdir(folderPath);
    end
    if ~isempty(dir(fullfile(fPath,fName,fName + "*.jp2")))
        delete(fullfile(fPath,fName,fName + "*.jp2"));
    end

    % Determine how many output files will be generated since the raw data
    % for compression cannot be larger than 2^32 - 1 bytes
    nPixV = size(imgs,1); nPixH = size(imgs,2); nImgs = size(imgs,3);
    pixPerImg = nPixV*nPixH;
    bytesPerImg = pixPerImg*2;
    nImgsPer4Gb = floor((2^32-1)/bytesPerImg);
    nOutputFiles = ceil(nImgs/nImgsPer4Gb);
    nImgsPerFile = ceil(nImgs/nOutputFiles);
    imgsInEachFile = min((1:nOutputFiles)*nImgsPerFile,nImgs) - (((1:nOutputFiles)-1)*nImgsPerFile);

    splitVals = [0 cumsum(imgsInEachFile)];

    % Save each chunk of images
    for i = 1:numel(splitVals)-1
        img = reshape(imgs(:,:,splitVals(i)+1:splitVals(i+1)),pixPerImg,imgsInEachFile(i));

        % This section is required for now because MATLAB crashes while
        % reading jp2 images of certain size, e.g. 16384x100000. Following code
        % will prove this. If MATLAB doesn't crash while running following code
        % then the bug is probably resolved. In that case delete this section
        % since it won't be necessary to do this anymore.
        %     img = zeros(100000,16384,"uint16");
        %     imwrite(img,"test.jp2","Mode","lossless");
        %     imgComp = imread("test.jp2");
        %
        % Find appropriate height and width for each cell element
        nPix = numel(img);
        % Can be done in two ways
        % 1. Try to make number of rows a multiple of pixPerImg and fit
```

```
        % multiple images in one column, gives better compression but may
        % cause crash
        nRows = ceil(sqrt(nPix)/pixPerImg)*pixPerImg;
        while mod(nPix,nRows)
            nRows = nRows + pixPerImg;
        end
        nCols = nPix/nRows;
        % 2. Aim for number of rows and number of columns being as close as
        % possible, will avoid crash better but provides poor compression
        nRows = ceil(sqrt(nPix));
        while mod(nPix,nRows)
            nRows = nRows + 1;
        end
        nCols = nPix/nRows;
        img = reshape(img,nRows,nCols);

        % Save the image
        imwrite(img,fullfile(fPath,fName,fName + num2str(i) + ".jp2"),"Mode","lossless",...
        ↪  Define height and width of the original images for future decoding.
                "Comment",[num2str(nPixV) num2str(nPixH) "First value is height and second
                ↪  value is the width of the original images."]);
    end

    % Reload the images to check if the files were saved correctly
    img = readImgsJp2(folderPath);

    % Delete bin file if the data read from jp2 files is same as the binary data
    if all(imgs == img,"all")
        delete(filePath);
    end
end
end
```

I.2.3 bin2mj2.m

```
function bin2mj2(filePath)
arguments
    filePath {mustBeFile}
end

% Load the images from binary file
imgs = readImgsBin(filePath);

if ~isempty(imgs)
    % Create a file with mj2 extension
    [fPath,fName,~] = fileparts(filePath);
    file = fullfile(fPath,fName) + ".mj2";
    v = VideoWriter(file,"Motion JPEG 2000");
    v.LosslessCompression = true;
```

```

    % Write frames to the file
    imgs = reshape(imgs,size(imgs,1),size(imgs,2),1,size(imgs,3));
    open(v);
    writeVideo(v,imgs);
    close(v);

    % Load the video frames back
    v = VideoReader(file);
    imgsComp = read(v);

    % Delete bin file if the data read from mj2 file is same as the binary data
    if all(imgs == imgsComp,"all")
        delete(filePath);
    else
        warning("Data from the compressed file doesn't seem to be the same as the bin
        ↪  file.\n");
    end
end
end
```

I.2.4 jp2ToAvi.m

```
function jp2ToAvi(jp2FolderPath,slowness,videoFilePath)
% This function assumes that the camera is capturing in the center of the
% sensor. If this is not the case then the calculation of the exposure time
% will be wrong.
arguments
    jp2FolderPath {mustBeFolder}
    slowness (1,1) double = 1
    videoFilePath (1,1) string = "video"
end

% Read jp2 files
imgs = readImgsJp2(jp2FolderPath);

% Adjust the contrast
imgs = imadjustn(imgs,stretchlim(imgs(:),0));

% Create an output file
[fPath,fName,~] = fileparts(videoFilePath);
vOut = VideoWriter(fullfile(fPath,fName),"Grayscale AVI");
vOut.FrameRate = 60;
open(vOut);

expTime = size(imgs,1)/2*9.74e-6; % Taken from camera datasheet
for i = 1:max(ceil(1/expTime/60/slowness),1):size(imgs,3)
    writeVideo(vOut,im2uint8(imgs(:,:,i)));
end
```

```

close(vOut);

end
```

I.2.5 mj2ToAvi.m

```
function mj2ToAvi(videoFilePath,expTime,slowness)
% This function assumes that the camera is capturing in the center of the
% sensor. If this is not the case then the calculation of the exposure time
% will be wrong.
arguments
    videoFilePath {mustBeFile}
    expTime (1,1) double
    slowness (1,1) double = 1
end

% Read mj2 file
imgs = readImgsMj2(videoFilePath);

% Adjust the contrast
imgs = imadjustn(imgs,stretchlim(imgs(:),0));

% Create an output file
[fPath,fName,~] = fileparts(videoFilePath);
vOut = VideoWriter(fullfile(fPath,fName),"Grayscale AVI");
vOut.FrameRate = 60;
open(vOut);

for i = 1:max(ceil(1/expTime/60/slowness),1):size(imgs,3)
    % imagesc(imgs(:,:,i));
    % colormap gray; axis equal; axis([-inf inf -inf inf])
    % writeVideo(vOut,im2uint8(rgb2gray(getframe(gcf).cdata)));
    writeVideo(vOut,im2uint8(imgs(:,:,i)));
end

close(vOut);

end
```

I.2.6 readImgsBin.m

```
function imgs = readImgsBin(filePath)

arguments
    filePath {mustBeFile}
end
```

```matlab

% Load the binary file containing images
imgsFile = fopen(filePath,"r");
if imgsFile > 2
    % Read img size and number of frames
    imgWidth = fread(imgsFile,1,"uint32"); % This was 'uint16' previously
    imgHeight = fread(imgsFile,1,"uint32"); % This was 'uint16' previously
    nImgs = fread(imgsFile,1,"uint32"); % This was 'uint16' previously

    % Check if number of frames is correct
    s = dir(filePath);
    filesize = s.bytes;
    nImgsComp = (filesize - 12)/double(imgWidth*imgHeight*2);

    if mod(filesize,1) ~= 0
        warning("Invalid image file.");
        imgs = [];
    else
        try
            % Make sure we have correct number of images
            if nImgsComp ~= nImgs
                disp("Mismatch between the number of frames stated in the file, "...
                       + num2str(nImgs)...
                       + ", vs what is available in the file, "...
                       + num2str(nImgsComp) + ".");
                nImgs = nImgsComp;
                % nImgs = input("Please enter the correct number of frames: ");
            end

            % Allocate memory to store images
            imgs = zeros(imgHeight,imgWidth,nImgs,"uint16");

            % Read images
            for i = 1:nImgs
                imgs(:,:,i) = fread(imgsFile,[imgWidth imgHeight],"uint16")';
            end

            % Close the file
            fclose(imgsFile);
        catch
            warning("Error reading the file.");
            imgs = [];
        end
    end
end

end
```

I.2.7 readImgsJp2.m

```
function imgs = readImgsJp2(folderPath)

arguments
    folderPath {mustBeFolder}
end

% Get list of .jp2 files in the directory
files = dir(fullfile(folderPath,"*.jp2"));

% Check if there are any jp2 files
if isempty(files)
    error("Folder " + folderPath + " doesn't contain any jp2 files.");
end

% Make sure all the files are correct
imInfo = imfinfo(fullfile(files(1).folder,files(1).name));
nPixV = str2double(imInfo.Comments(1));
nPixH = str2double(imInfo.Comments(2));
nImgs = zeros(numel(files),2);
if mod(imInfo.Height*imInfo.Width,nPixH*nPixV)
    error("File " + fullfile(files(1).folder,files(1).name) + " seems to be currupted.");
end
nImgs(1,1) = 1; nImgs(1,2) = imInfo.Height*imInfo.Width/nPixH/nPixV;
for i = 2:numel(files)
    imInfo = imfinfo(fullfile(files(i).folder,files(i).name));
    if str2double(imInfo.Comments(1)) ~= nPixV || str2double(imInfo.Comments(2)) ~= nPixH
        error("Size of the images isn't consistent.");
    elseif mod(imInfo.Height*imInfo.Width,str2double(imInfo.Comments(1))*str2double(imInfo.Comments(2)))
        error("File " + fullfile(files(1).folder,files(1).name) + " seems to be currupted.");
    end

    % Count the total number of images
    nImgs(i,1) = 1 + nImgs(i-1,2);
    nImgs(i,2) = nImgs(i-1,2) + imInfo.Height*imInfo.Width/nPixH/nPixV;
end

% Create a matrix to hold the images
imgs = zeros(nPixV,nPixH,nImgs(end,2),"uint16");

% Load all the images
for i = 1:numel(files)
    imgs(:,:,nImgs(i,1):nImgs(i,2)) = reshape(imread(fullfile(files(i).folder,files(i).name)),nPixV,nPixH,[]);
end

end
```

I.2.8 readImgsMj2.m

```
function imgs = readImgsMj2(filePath)
arguments
    filePath {mustBeFile}
end

% Load the video frames
v = VideoReader(filePath);
imgs = read(v);

% Reduce the one extra dimension
imgs = reshape(imgs,size(imgs,1),size(imgs,2),size(imgs,4));

end
```

APPENDIX J

Code Listings for Processing Experimental Data

J.1 Directory tree

/
- `imBinarize/` .... Stores results of imBinarize approach
- `kernelFit/` .... Stores results of kernelFit approach
- `private/` .... Acts as a local library of functions
  - `compPSD.m` .... Calculates PSD profile
  - `cropImgs.m` .... Used to extract kernel image
  - `filtPSD.m` .... Filters the PSD profile
  - `findBeadCentroids.m` .... Computes particle centroids from images
  - `fitHydroFaxenPSD.m` .... Fits theoretical PSD to experimental data
- `brownian0.mj2` .... Compressed recording of Brownian motion experiment
- `brownian0.m` .... Main code to process Brownian motion data
- `traj0.mj2` .... Compressed recording of TRR experiment
- `traj0.m` .... Main code to process TRR data
- `README.md` .... Keeps the information regarding the experiments

J.2 Code Listings

J.2.1 `compPSD.m`

```matlab
function [psd,frq] = compPSD(pos,dt)
arguments
    pos (:,1) double
    dt (1,1) double
end

% Compute common values
Fs = 1/dt;
L = length(pos);
frq = 0:Fs/L:Fs/2;

% Take the mean position out of the data
pos = pos - mean(pos);

% Calculate PSD
dft = fft(pos);
dft = dft(1:L/2+1);
psd = (1/(Fs*L)) * abs(dft).^2;
psd(2:end-1) = 2*psd(2:end-1);
end
```

J.2.2 cropImgs.m

```
function imgs = cropImgs(imgs)
arguments
    imgs (:,:,:) uint16
end

imgsSz = size(imgs(:,:,1));

% Show the first image
imagesc(imgs(:,:,1));
colormap gray;
axis equal;
axis([-inf inf -inf inf]);

% Ask the user to enter the cropping region
cropRect = input("Enter xLow, xHigh, yLow and yHigh values of the crop region in array
↪  format: ");
close(imgcf);

if isequal(size(cropRect(:)), [4,1])
        cropRect(1) = max(cropRect(1),1);
        cropRect(2) = min(cropRect(2),imgsSz(2));
        cropRect(3) = max(cropRect(3),1);
        cropRect(4) = min(cropRect(4),imgsSz(1));
        cropRect = round(cropRect);
else
    cropRect = [1 imgsSz(2) 1 imgsSz(1)];
end

imgs = imgs(cropRect(3):cropRect(4),cropRect(1):cropRect(2),:);
end
```

J.2.3 filtPSD.m

```
function [psd,frq] = filtPSD(psd,frq,nValsPerBin)
arguments
    psd (:,1) double
    frq (:,1) double
    nValsPerBin (1,1) double
end

if length(psd) ~= length(frq)
    error("Length of psd and frq array don't match.");
end

[frqBins,frqEdges] = discretize(frq,ceil(length(frq)/nValsPerBin));
psd = accumarray(frqBins(:),psd(:))./accumarray(frqBins(:),1);
```

```
frq = (frqEdges(1:end-1)+frqEdges(2:end))'./2;
end
```

J.2.4 findBeadCentroids.m

```
function [xc,yc] = findBeadCentroids(imgs,method,beadImg,threshold)

arguments
    imgs (:,:,:) uint16
    method (1,:) char {mustBeMember(method,{'kernelFit','imBinarize'})}
    beadImg (:,:) uint16 = []
    threshold (1,1) double {mustBeInRange(threshold,0,1)} = 0.6
end

%% Get some dimensions
imgsSz = size(imgs(:,:,1));
nImgs = size(imgs,3);

%% Extract the bead image if not provided
if isempty(beadImg)
    % Show the second image, sometimes the first image has issues
    if nImgs > 1
        img = imgs(:,:,2);
    else
        img = imgs(:,:,1);
    end
    imagesc(img);
    colormap gray;
    axis equal;
    axis([-inf inf -inf inf]);

    % Ask the user to enter the cropping region
    cropRect = input("Enter xLow, xHigh, yLow and yHigh values of the bead region in array format: ");
    close(imgcf);
    if isequal(size(cropRect(:)), [4,1])
        cropRect(1) = max(cropRect(1),1);
        cropRect(2) = min(cropRect(2),imgsSz(2));
        cropRect(3) = max(cropRect(3),1);
        cropRect(4) = min(cropRect(4),imgsSz(1));
        cropRect = round(cropRect);
    else
        cropRect = [1 imgsSz(2) 1 imgsSz(1)];
    end

    beadImg = img(cropRect(3):cropRect(4),cropRect(1):cropRect(2));
end

%% Preallocate arrays
```

```
xc = zeros(nImgs,1);
yc = zeros(nImgs,1);

%% Find bead centroids
if isequal(method,"kernelFit")
    % Scale the kernel
    beadImg = double(beadImg);
    beadImg = beadImg - mean(beadImg,"all");
    beadSz = size(beadImg);

    % Compute the cross-correlation value
    % taken from: Tracking kinesin-driven movements with nanometre-scale precision
    parfor i = 1:nImgs
        img = im2double(imgs(:,:,i));
        img = img - mean(img,"all");

        C = xcorr2(img,beadImg);
        C = C((1:imgsSz(1)) + (beadSz(1) - 1)/2, (1:imgsSz(2)) + (beadSz(2) - 1)/2);
        C = C./max(C,[],"all");

        % Compute the centroid
        CminThr = C - threshold;
        CminThr(CminThr < 0) = 0;
        CminThrSum = sum(CminThr,"all");
        r = 1:imgsSz(1); c = 1:imgsSz(2);
        xc(i) = sum(CminThr.*c,"all")/CminThrSum; yc(i) =
        ↪  sum(CminThr.*r',"all")/CminThrSum;
    end
elseif isequal(method,"imBinarize")
    % Find area of the bead in beadImg
    beadImg = wiener2(beadImg,[3 3]);
    beadImg = imbinarize(beadImg);
    [labeled,~] = bwlabel(beadImg,8);
    graindata = regionprops(labeled,'basic');
    beadArea = max([graindata.Area]);

    % Use 60% of the beadArea to find the bead in all the images
    parfor i = 1:nImgs
        img = imgs(:,:,i);
        img = wiener2(img,[3 3]);
        bw = imbinarize(img);
        [labeled,~] = bwlabel(bw,4);
        graindata = regionprops(labeled,'basic');
        graindata = graindata([graindata.Area] > threshold*beadArea);
        maxArea = min([graindata.Area]);
        biggestGrain = [graindata.Area]==maxArea;
        xc(i) = graindata(biggestGrain).Centroid(1);
        yc(i) = graindata(biggestGrain).Centroid(2);
    end
```

```
end
end
```

J.2.5 fitHydroFaxenPSD.m

```
function [D,k,convFact,fHndl] = fitHydroFaxenPSD(psd,frq,beadDia,wallDistance,beadDensity,dynViscosity,frqMin,frqMax)
arguments
    psd (:,1) double
    frq (:,1) double
    beadDia (1,1) double
    wallDistance (1,1) double
    beadDensity (1,1) double
    dynViscosity (1,1) double
    frqMin (1,1) double
    frqMax (1,1) double
end

%% All parameters values are assumed to be defined with unit system kgram, meter, second. %%
temp = 293.15; % System Temperature in kelvin
k_B = 1.38064852*1e-23; % Boltzmann Constant in meter^2*kgram/second^2/kelvin
rho_b = beadDensity; % Bead density in kg/m^3
rho_m = 1000; % Density of water in kg/m^3
r = beadDia/2; % Bead radius in meter
Vol = 4/3*pi*r*r*r; % Bead volume in m^3
m = rho_b*Vol; % Bead mass in kg
m_star = m + 2*pi*r*r*r*rho_m; % Hydrodynamical mass
mu_m = dynViscosity; % Dynamic viscosity of fluid medium at 20d C in Ns/m^2
nu_m = mu_m/rho_m; % Kinematic viscosity of water
beta_v = 6*pi*mu_m*r; % Translational drag coefficient Ns/m

fv = nu_m/pi/r/r; % Frequency where delta = R
fm = beta_v/2/pi/m_star; % Frequency of inertial relaxation
rlRatio = r/wallDistance; % Used for Faxen's drag coefficient correction

%% Try to compute k and D using optimization %%
% Limit the frequency range
idx = frq >= frqMin & frq <= frqMax; frq = log10(frq(idx)); psd = log10(psd(idx)); % Use log scaled values

% Set strict convergence criteria
opt = optimoptions('lsqcurvefit','Algorithm','levenberg-marquardt','Display','none',...
                   'OptimalityTolerance',1e-100,'FunctionTolerance',1e-100,...
                   'StepTolerance',1e-300,'MaxFunctionEvaluations',10000);

% Fit the function
x0 = [1e-4 100]; % [D fc]
x = lsqcurvefit(@(x,f) psdFunc(x,f,fv,fm,rlRatio),x0,frq,psd,[0 frqMin],[inf frqMax],opt);
```

```matlab

% Display the result
D = x(1); fc = x(2);
k = fc*2*pi*beta_v;
convFact = sqrt(k_B*temp/beta_v/D);
fHndl = @(f) D/2/pi^2./(fc^2 + (1 - 4*pi*fc*eps)*f.^2 + 4*pi^2*eps^2*f.^4);
disp("Conversion factor is " + num2str(sqrt(k_B*temp/beta_v/D)*1e6) + "um/px and stiffness
↪  is " + num2str(k*1e6) + "pN/um.");

end

function psd = psdFunc(x,f,fv,fm,rlRatio)
D = x(1); fc = x(2);
f = 10.^f; % Because the input is log scaled
reGmGm0 = 1 + sqrt(f./fv) - 3*rlRatio/16 +
↪  3*rlRatio/4*exp(-2/rlRatio*sqrt(f./fv)).*cos(2/rlRatio*sqrt(f./fv));
imGmGm0 = - sqrt(f./fv) +
↪  3*rlRatio/4*exp(-2/rlRatio*sqrt(f./fv)).*sin(2/rlRatio*sqrt(f./fv));
psd = log10(D.*reGmGm0 / (2*pi^2) ./ ( (fc + f.*imGmGm0 - (f.^2)./fm).^2 + (f.*reGmGm0).^2
↪  ) );
end
```

J.2.6 brownian0.m

```matlab
clear; close all; clc;

%% Define necessary variables
fileName = "brownian0";
imProcessingMethod = "kernelFit";

%% Load the images
imgs = readImgsMj2(fileName + ".mj2");
intensityLims = stretchlim(imgs(:),0);

% Adjust images for contrast
if isequal(imProcessingMethod,"kernelFit")
    imgs = imadjustn(imgs,intensityLims);
elseif isequal(imProcessingMethod,"imBinarize")
    imgs = imadjustn(imgs,intensityLims);
end

%% Crop the images
imgs = cropImgs(imgs);

%% Extract image centers
if isequal(imProcessingMethod,"kernelFit")
    [xc,yc] = findBeadCentroids(imgs,imProcessingMethod); % kernelFit: [17 43 11 37]
elseif isequal(imProcessingMethod,"imBinarize")
    [xc,yc] = findBeadCentroids(imgs,imProcessingMethod,[],0.4);
```

```
end

% Plot the image with the detected centroid
im = imagesc(imgs(:,:,1));
colormap gray; axis equal; axis([-inf inf -inf inf])
hold on;
s = scatter(xc(1,:),yc(1,:),"r*");
hold off;
for i = 1:size(imgs,3)
    im.CData = imgs(:,:,i);
    s.XData = xc(i,:); s.YData = yc(i,:);
    drawnow limitrate;
end
close(gcf);

%% Create time array and store the information
dt = 0.00025; % Taken from README.md
time = (0:size(imgs,3)-1)*dt;
mkdir(imProcessingMethod);
save(imProcessingMethod + "\" + fileName + ".mat","time","xc","yc");

%% Compute power spectrum %%
xc = xc(1:125000); yc = yc(1:125000); time = time(1:125000);
% Make the number of datapoints even
if rem(length(time),2) ~= 0
    xc(end) = []; yc(end) = []; time(end) = [];
end

[PSD_X,FRQ_X] = compPSD(xc,time(2));
[PSD_Y,FRQ_Y] = compPSD(yc,time(2));

%% Post-Process the PSD Data %%
% Cut the data to start from 1Hz frequency
PSD_X = PSD_X(FRQ_X>=1); FRQ_X = FRQ_X(FRQ_X>=1);
PSD_Y = PSD_Y(FRQ_Y>=1); FRQ_Y = FRQ_Y(FRQ_Y>=1);

% Put the data into bins and average out
[psdX,frqX] = filtPSD(PSD_X,FRQ_X,30);
[psdY,frqY] = filtPSD(PSD_Y,FRQ_Y,30);

%% Plot the power spectrum %%
% For x direction
figure();
loglog(frqX,psdX);
grid on
title("PSD for x-direction")

% For y direction
figure();
loglog(frqY,psdY);
grid on
```

```
title("PSD for y-direction")

%% Fit PSD data
beadDia = 0.5e-6;
beadDensity = 1060;
waterMu = 1.0016e-3;
[DX,kX,convFactX,fHndlX] = fitHydroFaxenPSD(psdX,frqX,beadDia,10e-6,beadDensity,waterMu,50,1000);
[DY,kY,convFactY,fHndlY] = fitHydroFaxenPSD(psdY,frqY,beadDia,10e-6,beadDensity,waterMu,50,1000);
save(imProcessingMethod + "\" + fileName + ".mat","DX","kX","convFactX","fHndlX","DY","kY","convFactY","fHndlY","-append");

%% Plot the curvefit and save the figures
figure(1);
hold on;
loglog(frqX,fHndlX(frqX),'r',"LineWidth",1);
hold off

figure(2);
hold on;
loglog(frqY,fHndlY(frqY),'r',"LineWidth",1);
hold off
```

J.2.7 traj0.m

```
clear; close all; clc;

%% Define necessary variables
fileName = "traj0";
imProcessingMethod = "kernelFit";

%% Load the images
imgs = readImgsMj2(fileName + ".mj2");
intensityLims = stretchlim(imgs(:),0);

% Adjust images for contrast
if isequal(imProcessingMethod,"kernelFit")
    imgs = imadjustn(imgs,intensityLims);
elseif isequal(imProcessingMethod,"imBinarize")
    imgs = imadjustn(imgs,intensityLims);
end

%% Crop the images
imgs = cropImgs(imgs);

%% Extract image centers
if isequal(imProcessingMethod,"kernelFit")
    [xc,yc] = findBeadCentroids(imgs,imProcessingMethod); % kernelFit: [23 43 14 34]
```

```
elseif isequal(imProcessingMethod,"imBinarize")
    [xc,yc] = findBeadCentroids(imgs,imProcessingMethod,[],0.4);
end
% Plot the image with the detected centroid
im = imagesc(imgs(:,:,1));
colormap gray; axis equal; axis([-inf inf -inf inf])
hold on;
s = scatter(xc(1,:),yc(1,:),"r*");
hold off;
for i = 1:size(imgs,3)
    im.CData = imgs(:,:,i);
    s.XData = xc(i,:); s.YData = yc(i,:);
    drawnow limitrate;
    pause(0.1);
end
close(gcf);

%% Create time array and store the information
dt = 0.00032; % Taken from README.md
time = (0:size(imgs,3)-1)*dt;
save(imProcessingMethod + "\" + fileName + ".mat","time","xc","yc");

%% Generate plots and save them
mkdir(imProcessingMethod + "\figs\");
figure("WindowState","maximized");
t = tiledlayout(2,2,"TileSpacing","compact","Padding","compact");

xData = nexttile(1);
plot(time,xc*65);
grid on;
axis([-inf inf -inf inf]);
xlabel("Time (s)");
ylabel("x-Displacement (nm)");

yData = nexttile(3);
plot(time,yc*65);
grid on;
axis([-inf inf -inf inf]);
xlabel("Time (s)");
ylabel("y-Displacement (nm)");

xyData = nexttile(2,[2 1]);
plot(xc*65,yc*65);
grid on;
axis equal;
xlabel("x-Displacement (nm)");
ylabel("y-Displacement (nm)");

% Link time axes of the plots
linkaxes([xData yData],'x');

```

```
% Save the figure
savefig(imProcessingMethod + "\figs\" + fileName + ".fig");
exportgraphics(gcf,imProcessingMethod + "\figs\" + fileName + ".jpg","Resolution",600);
```

J.2.8 README.md

```
## Brownian motion and trapping motion data for 500nm bead
### Setup Info
- Number of Samples: 1
  - Sample 1:
    - Bead Diameter: 500nm
    - Objective Zoom: 100X
    - Objective NA: 1.25
    - Centrifuged for: 0 min
    - Optical Table: On
    - Number of ND filters: 1 thorlabs
  - Sample 2:
    - Bead Diameter: 500nm
    - Objective Zoom: 100X
    - Objective NA: 1.25
    - Centrifuged for: 0 min
    - Optical Table: On
    - Number of ND filters: 1 thorlabs + ND4 from uScope

### Trapping Set 0
Sample: 1
Particle: 1

#### Camera configuration
`camProp setByName "SUBARRAY MODE" 2`
`camProp setByName "SUBARRAY VSIZE" 40`
`camProp setByName "SUBARRAY VPOS" 1004`
`camProp setByName "SUBARRAY HSIZE" 64`
`camProp setByName "SUBARRAY HPOS" 796`
`camProp setByName "EXPOSURE TIME" 0.00032`
`shtrCtrl "led 30"`
`liveCap start`
`liveCap lut 100 200`
`liveCap stop`

#### Recording
`camProp setByName "SENSOR COOLER" 2`
`shtrCtrl "cycleTiming 30000 20000 100000"`
`shtrCtrl "led 100"`
`condExps start 1`

### Brownian Set 0
Sample: 2
Particle: 2
```

```

#### Camera configuration
`camProp setByName "SUBARRAY MODE" 2`
`camProp setByName "SUBARRAY VSIZE" 48`
`camProp setByName "SUBARRAY VPOS" 1000`
`camProp setByName "SUBARRAY HSIZE" 64`
`camProp setByName "SUBARRAY HPOS" 796`
`camProp setByName "EXPOSURE TIME" 0.00025`
`shtrCtrl "led 30"`
`liveCap start`
`liveCap lut 100 200`
`liveCap stop`

#### Recording
`shtrCtrl "led 100"`
`camProp setByName "SENSOR COOLER" 2`
`camRec 250000 brownian0.bin`

#### Camera Reset Configuration (Let the system be in this state for camera and illumination laser to cool down)
`shtrCtrl "led 30"`
`camProp setByName "SUBARRAY MODE" 1`
`camProp setByName "EXPOSURE TIME" 0.01`
`camProp setByName "SENSOR COOLER" 1`
`liveCap start`
`liveCap lut 100 200`
`liveCap stop`

#### Shutter Commands
`shtrCtrl "shtrClose"`
`shtrCtrl "shtrOpen"`
```

REFERENCES

[1] Jonghoon Ahn, Zhujing Xu, Jaehoon Bang, Yu-Hao Deng, Thai M. Hoang, Qinkai Han, Ren-Min Ma, and Tongcang Li. Optically levitated nanodumbbell torsion Balance and GHz nanomechanical rotor. *Physical Review Letters*, 121:033603, Jul 2018.

[2] Davide Andreis and Enrico Canuto. Orbit dynamics and kinematics with full quaternions. *IFAC Proceedings Volumes*, 37(6):185–190, 2004. 16th IFAC Symposium on Automatic Control in Aerospace 2004, Saint-Petersburg, Russia, 14-18 June 2004.

[3] Michael S. Andrle and John L. Crassidis. Geometric integration of quaternions. *Journal of Guidance, Control, and Dynamics*, 36(6):1762–1767, 2013.

[4] A. Ashkin. Forces of a single-beam gradient laser trap on a dielectric sphere in the ray optics regime. *Biophysical journal*, 61(2):569–582, Feb 1992.

[5] Arthur Ashkin. Optical trapping and manipulation of neutral particles using lasers. *Proceedings of the National Academy of Sciences*, 94(10):4853–4860, 10 1997.

[6] Iddo Ben-Ari, Khalid Boushaba, Anastasios Matzavinos, and Alexander Roitershtein. Stochastic analysis of the motion of dna nanomechanical bipeds. *Bulletin of Mathematical Biology*, 73(8):1932–1951, November 2010.

[7] Julius S. Bendat and Allan G. Piersol. *Random Data: Analysis and Measurement Procedures*. Wiley, January 2010.

[8] Kirstine Berg-Sørensen and Henrik Flyvbjerg. Power spectrum analysis for optical tweezers. *Review of Scientific Instruments*, 75(3):594–612, 2004. [doi:10.1063/1.1645654].

[9] Kirstine Berg-Sørensen, Erwin J. G. Peterman, Tom Weber, Christoph F. Schmidt, and Henrik Flyvbjerg. Power spectrum analysis for optical tweezers. ii: Laser wavelength dependence of parasitic filtering, and how to achieve high bandwidth. *Review of Scientific Instruments*, 77(6):063106, 2006. [doi:10.1063/1.2204589].

[10] Johannes Berner, Boris Müller, Juan Ruben Gomez-Solano, Matthias Krüger, and Clemens Bechinger. Oscillating modes of driven colloids in overdamped systems. *Nature Communications*, 9(1):999, 2018.

[11] Peter Betsch. On the use of Euler parameters in multibody dynamics. *PAMM*, 6(1):85–86, 2006.

[12] Peter Betsch and Ralf Siebert. Rigid body dynamics in terms of quaternions: Hamiltonian formulation and conserving numerical integration. *International Journal for Numerical Methods in Engineering*, 79(4):444–473, 2009.

[13] Anna S. Bodrova, Aleksei V. Chechkin, Andrey G. Cherstvy, Hadiseh Safdari, Igor M. Sokolov, and Ralf Metzler. Underdamped scaled brownian motion: (non-)existence of the overdamped limit in anomalous diffusion. *Scientific Reports*, 6(1):30520, Jul 2016. [doi:10.1038/srep30520].

[14] Sebastián Bouzat and Fernando Falo. The influence of direct motor-motor interaction in models for cargo transport by a single team of motors. *Physical Biology*, 7(4):046009, November 2010.

[15] A. Bowling. *Vector Mechanics: A Systematic Approach*. Aqualan Press, LLC, 2018.

[16] Alan Bowling, Vatsal Joshi, and Mahdi Haghshenas-Jaryani. Modeling and experimental exploration of the underdamped motion of microbeads in optical tweezers. In Kishan Dholakia and Gabriel C. Spalding, editors, *Optical Trapping and Optical Micromanipulation XVI*, volume 11083, pages 92 – 107. International Society for Optics and Photonics, SPIE, September 2019. [doi:10.1117/12.2530017].

[17] Richard Bowman, Graham Gibson, and Miles Padgett. Particle tracking stereomicroscopy in optical tweezers: Control of trap shape. *Opt. Express*, 18(11):11785–11790, May 2010.

[18] J.C. Butcher. A history of runge-kutta methods. *Applied Numerical Mathematics*, 20(3):247–260, March 1996.

[19] Frederic Català, Ferran Marsà, Mario Montes-Usategui, Arnau Farré, and Estela Martín-Badosa. Influence of experimental parameters on the laser heating of an optical trap. *Scientific Reports*, 7(1), November 2017.

[20] Jim C. Chen and Albert S. Kim. Brownian dynamics, molecular dynamics, and monte carlo modeling of colloidal systems. *Advances in Colloid and Interface Science*, 112(1-3):159–173, December 2004.

[21] J. C. K. Chou. Quaternion kinematic and dynamic differential equations. *IEEE Transactions on Robotics and Automation*, 8(1):53–64, Feb 1992.

[22] A. Ciudad, J. M. Sancho, and G. P. Tsironis. Kinesin as an electrostatic machine. *Journal of Biological Physics*, 32(5):455–463, December 2006.

[23] P. E. Crouch and R. Grossman. Numerical integration of ordinary differential equations on manifolds. *Journal of Nonlinear Science*, 3(1):1–33, Dec 1993.

[24] Subhasish Dey, Sk Zeeshan Ali, and Ellora Padhi. Terminal fall velocity: the legacy of stokes from the perspective of fluvial hydraulics. *Proceedings*

of the Royal Society A: Mathematical, Physical and Engineering Sciences, 475(2228):20190277, August 2019.

[25] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. https://www.astro.rug.nl/software/kapteyn-beta/_downloads/attitude.pdf, 2006. [Online; accessed 10-Jan-2022].

[26] Mohammad Farazmand and George Haller. The maxey–riley equation: Existence, uniqueness and regularity of solutions. *Nonlinear Analysis: Real World Applications*, 22:98–106, April 2015.

[27] Thomas J. Fellers and Michael W. Davidson. Coverslip Correction — microscopyu.com. https://www.microscopyu.com/microscopy-basics/coverslip-correction. [Accessed 08-11-2023].

[28] Thomas Franosch, Matthias Grimm, Maxim Belushkin, Flavio M. Mor, Giuseppe Foffi, László Forró, and Sylvia Jeney. Resonances arising from hydrodynamic memory in brownian motion. *Nature*, 478(7367):85–88, October 2011.

[29] Jeff Gelles, Bruce J. Schnapp, and Michael P. Sheetz. Tracking kinesin-driven movements with nanometre-scale precision. *Nature*, 331(6155):450–453, February 1988.

[30] G. Gouesbet and G. Gréhan. *Generalized Lorenz-Mie Theories*. Springer International Publishing, 2 edition, 2017. [doi:10.1007/978-3-642-17194-9].

[31] Honglian Guo, Qinhong Cao, Dongtao Ren, Guoqing Liu, Jianfa Duan, Zhaolin Li, Daozhong Zhang, and Xuehai Han. Measurements of leucocyte membrane elasticity based on the optical tweezers. *Chinese Science Bulletin*, 48(5):503–508, Mar 2003.

[32] Gopal K. Gupta, Ron Sacks-Davis, and Peter E. Tescher. A review of recent developments in solving odes. *ACM Comput. Surv.*, 17(1):5–47, mar 1985. [doi:10.1145/4078.4079].

[33] Mahdi Haghshenas-Jaryani, Bryan James Black, Sarvenaz Ghaffari, James Drake, Alan Bowling, and Samarendra Mohanty. Dynamics of microscopic objects in optical tweezers: Experimental determination of underdamped regime and numerical simulation using multiscale analysis. *Nonlinear Dynamics*, 76(2):1013–1030, April 2014. [doi:10.1007/s11071-013-1185-0].

[34] Mahdi Haghshenas-Jaryani and Alan Bowling. A new switching strategy for addressing euler parameters in dynamic modeling and simulation of rigid multibody systems. *Multibody System Dynamics*, 30(2):185–197, Aug 2013.

[35] Iddo Heller, Tjalle P. Hoekstra, Graeme A. King, Erwin J. G. Peterman, and Gijs J. L. Wuite. Optical tweezers analysis of dna-protein complexes. *Chemical Reviews*, 114(6):3087–3119, 2014.

[36] Desmond J. Higham. An algorithmic introduction to numerical simulation of stochastic differential equations. *SIAM Review*, 43(3):525–546, 2001. [doi:10.1137/S0036144500378302].

[37] An Ho, Alan George, and Ann Gordon-Ross. Improving compression ratios for high bit-depth grayscale video formats. In *2016 IEEE Aerospace Conference*. IEEE, March 2016.

[38] David Hoag. Apollo guidance and navigation: Considerations of apollo imu gimbal lock. *Apollo Lunar Surface Journal*, 1963.

[39] Yaozhong Hu. Semi-implicit euler-maruyama scheme for stiff stochastic equations. In H. Körezlioğlu, B. Øksendal, and A. S. Üstünel, editors, *Stochastic Analysis and Related Topics V*, pages 183–202, Boston, MA, 1996. Birkhäuser Boston.

[40] John E. Hurtado and Andrew J. Sinclair. Lagrangian mechanics of overparameterized systems. *Nonlinear Dynamics*, 66(1):201–212, Oct 2011.

[41] Kiyosi Itô. Stochastic integral. *Proceedings of the Imperial Academy*, 20(8):519 – 524, 1944. [doi:10.3792/pia/1195572786].

[42] Vatsal Joshi, Alan Bowling, and Mahdi Haghshenas-Jaryani. Modeling and experimental exploration of the underdamped motion of microbeads in optical tweezers. In Kishan Dholakia and Gabriel C. Spalding, editors, *Optical Trapping and Optical Micromanipulation XVI*. SPIE, September 2019.

[43] Vatsal Joshi and Alan P. Bowling. Investigation of the power spectral density of a scaled model simulation of an optical tweezer. In Kishan Dholakia and Gabriel C. Spalding, editors, *Optical Trapping and Optical Micromanipulation XIX*. SPIE, October 2022.

[44] Yonggun Jun, Suvranta K. Tripathy, Babu R.J. Narayanareddy, Michelle K. Mattson-Hoss, and Steven P. Gross. Calibration of optical tweezers for in vivo force measurements: How do different approaches compare? *Biophysical Journal*, 107(6):1474–1484, September 2014.

[45] Thomas R. Kane and David A. Levinson. *Dynamics: Theory and Applications*. McGraw Hill, 1985.

[46] Charles F.F. Karney. Quaternions in molecular modeling. *Journal of Molecular Graphics and Modelling*, 25(5):595–604, 2007.

[47] J. W. J. Kerssemakers, M. E. Janson, A. van der Horst, and M. Dogterom. Optical trap setup for measuring microtubule pushing forces. *Applied Physics Letters*, 83(21):4441–4443, 2003.

[48] Do-Nyun Kim, Cong-Tri Nguyen, and Mark Bathe. Conformational dynamics of supramolecular protein assemblies. *Journal of Structural Biology*, 173(2):261–270, February 2011.

[49] P. E. Kloeden and E. Platen. A survey of numerical methods for stochastic differential equations. *Stochastic Hydrology and Hydraulics*, 3(3):155–178, Sep 1989. [doi:10.1007/BF01543857].

[50] Ilya Konyshev and Andrey Byvalov. Model systems for optical trapping: the physical basis and biological applications. *Biophysical Reviews*, 13(4):515–529, July 2021.

[51] Elyahb A. Kwizera, Mingrui Sun, Alisa M. White, Jianrong Li, and Xiaoming He. Methods of generating dielectrophoretic force for microfluidic manipulation of bioparticles. *ACS Biomaterials Science & Engineering*, 7(6):2043–2063, April 2021.

[52] Woei Ming Lee, Peter J Reece, Robert F Marchington, Nikolaus K Metzger, and Kishan Dholakia. Construction and calibration of an optical trap on a fluorescence optical microscope. *Nature Protocols*, 2(12):3226–3238, December 2007.

[53] Isaac C. D. Lenton, Timo A. Nieminen, Vincent L. Y. Loke, Alexander B. Stilgoe, Y. Hu, Gregor Knöner, Agata M. Brańczyk, Norman R. Heckenberg, and Halina Rubinsztein-Dunlop. Optical tweezers toolbox. `https://github.com/ilent2/ott`, 2020.

[54] Zhisong Liu and Yueke Jia. Two simulation methods of brownian motion. *Journal of Physics: Conference Series*, 2012(1):012015, sep 2021. [doi:10.1088/1742-6596/2012/1/012015].

[55] F. Lizarralde and J. T. Wen. Attitude control without angular velocity measurement: a passivity approach. *IEEE Transactions on Automatic Control*, 41(3):468–472, March 1996.

[56] L. P. Maguire, S. Szilagyi, and R. E. Scholten. High performance laser shutter using a hard disk drive voice-coil actuator. *Review of Scientific Instruments*, 75(9):3077–3079, September 2004.

[57] Erik Meijering, Oleh Dzyubachyk, and Ihor Smal. Methods for cell and particle tracking. In P. Michael conn, editor, *Imaging and Spectroscopic Analysis of Living Cells - Optical and Spectroscopic Techniques*, pages 183–200. Elsevier, 2012.

[58] Michael Möller and Christoph Glocker. Rigid body dynamics with a scalable body, quaternions and perfect constraints. *Multibody System Dynamics*, 27(4):437–454, Apr 2012.

[59] Rebecca K. Montange, Matthew S. Bull, Elisabeth R. Shanblatt, and Thomas T. Perkins. Optimizing bead size reduces errors in force measurements in optical traps. *Opt. Express*, 21(1):39–48, Jan 2013.

[60] Hans Munthe-Kaas. Runge-Kutta methods on Lie groups. *BIT Numerical Mathematics*, 38(1):92–111, Mar 1998.

[61] A.H. Nayfeh. *Perturbation Methods*. Physics textbook. Wiley, 2008. [doi:10.1002/9783527617609].

[62] Keir C. Neuman and Steven M. Block. Optical trapping. *The Review of scientific instruments*, 75(9):2787–2809, Sep 2004. [doi:10.1063/1.1785844].

[63] Timo A. Nieminen, Gregor Knöner, Norman R. Heckenberg, and Halina Rubinsztein-Dunlop. Physics of optical tweezers. In *Methods in Cell Biology*, volume 9, pages 207–236. Elsevier, jul 2007. [doi:10.1088/1464-4258/9/8/s12].

[64] Timo A Nieminen, Vincent L Y Loke, Alexander B Stilgoe, Gregor Knöner, Agata M Brańczyk, Norman R Heckenberg, and Halina Rubinsztein-Dunlop. Optical tweezers computational toolbox. *Journal of Optics A: Pure and Applied Optics*, 9(8):S196–S203, jul 2007. [doi:10.1088/1464-4258/9/8/s12].

[65] Timo A. Nieminen, Vincent L.Y. Loke, Alexander B. Stilgoe, Norman R. Heckenberg, and Halina Rubinsztein-Dunlop. T-matrix method for modelling optical tweezers. *Journal of Modern Optics*, 58(5-6):528–544, 2011. [doi:10.1080/09500340.2010.528565].

[66] Nikon. CFI Achromat Series — microscope.healthcare.nikon.com. https://www.microscope.healthcare.nikon.com/products/optics/cfi-achromat-series. [Accessed 08-11-2023].

[67] P. E. Nikravesh, O. K. Kwon, and R. A. Wehage. Euler parameters in computational kinematics and dynamics. part 2. *Journal of Mechanisms, Transmissions, and Automation in Design*, 107(3):366–369, Sep 1985.

[68] P. E. Nikravesh, R. A. Wehage, and O. K. Kwon. Euler parameters in computational kinematics and dynamics. part 1. *Journal of Mechanisms, Transmissions, and Automation in Design*, 107(3):358–365, Sep 1985.

[69] Oliver M. O'Reilly and Peter C. Varadi. Hoberman's sphere, Euler parameters and Lagrange's equations. *Journal of Elasticity*, 56(2):171–180, Aug 1999.

[70] Giuseppe Pesce, Philip H. Jones, Onofrio M. Maragò, and Giovanni Volpe. Optical tweezers: theory and practice. *The European Physical Journal Plus*, 135(12):949, Dec 2020. [doi:10.1140/epjp/s13360-020-00843-5].

[71] A. Pralle, M. Prummer, E.-L. Florin, E.H.K. Stelzer, and J.K.H. Hörber. Three-dimensional high-resolution particle tracking for optical tweezers by forward scattered light. *Microscopy Research and Technique*, 44(5):378–386, March 1999.

[72] E. M. Purcell. Life at low reynolds number. *American Journal of Physics*, 45(1):3–11, 1977. [doi:10.1119/1.10903].

[73] Chris Rackauckas and Qing Nie. Stability-optimized high order methods and stiffness detection for pathwise stiff stochastic differential equations. In *2020*

IEEE High Performance Extreme Computing Conference (HPEC), pages 1–8, 2020. [doi:10.1109/HPEC43674.2020.9286178].

[74] H. Risken. *The Fokker-Planck Equation: Methods of Solution and Applications*. Springer Series in Synergetics. Springer Berlin Heidelberg, 2012.

[75] Alexander Rohrbach. Stiffness of optical traps: Quantitative agreement between experiment and electromagnetic theory. *Phys. Rev. Lett.*, 95:168102, Oct 2005.

[76] Yael Roichman, Alex Waldron, Emily Gardel, and David G. Grier. Optical traps with geometric aberrations. *Applied Optics*, 45(15):3425, May 2006.

[77] Felix Ruhnow, David Zwicker, and Stefan Diez. Tracking single particles and elongated filaments with nanometer precision. *Biophysical Journal*, 100(11):2820–2828, June 2011.

[78] Simo Särkkä and Arno Solin. *Applied Stochastic Differential Equations*. Cambridge University Press, April 2019.

[79] Mohammad Sarshar, Winson T. Wong, and Bahman Anvari. Comparative study of methods to calibrate the stiffness of a single-beam gradient-force optical tweezers over various laser trapping powers. *Journal of Biomedical Optics*, 19(11):115001, November 2014.

[80] Tamar Schlick. *Molecular Modeling and Simulation: An Interdisciplinary Guide*. Interdisciplinary Applied Mathematics 21. Springer-Verlag New York, 2 edition, 2010.

[81] Nur Adila Faruk Senan and Oliver M. O'Reilly. On the use of quaternions and Euler-Rodrigues symmetric parameters with moments and moment potentials. *International Journal of Engineering Science*, 47(4):595–609, 2009.

[82] Ahmed A Shabana. Euler parameters kinetic singularity. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 228(3):307–313, 2014.

[83] Lawrence F. Shampine and Mark W. Reichelt. The matlab ode suite. *SIAM J. Sci. Comput.*, 18(1):1–22, jan 1997. [doi:10.1137/S1064827594276424].

[84] Karim Sherif, Karin Nachbagauer, and Wolfgang Steiner. On the rotational equations of motion in rigid body dynamics when using Euler parameters. *Nonlinear Dynamics*, 81(1):343–352, Jul 2015.

[85] Ravishankar Shivarama and Eric P. Fahrenthold. Hamilton's equations with Euler parameters for rigid body dynamics modeling . *Journal of Dynamic Systems, Measurement, and Control*, 126(1):124–130, 04 2004.

[86] MD Shuster. A survey of attitude representations. *Journal of the Astronautical Sciences*, 41(4):439–517, 1993.

[87] R. M. Simmons, J. T. Finer, S. Chu, and J. A. Spudich. Quantitative measurements of force and displacement using an optical trap. *Biophysical journal*, 70(4):1813–1822, Apr 1996.

[88] J.W. Strutt. XV. on the light from the sky, its polarization and colour. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(271):107–120, February 1871.

[89] Aksel Sveier, Alexander M. Sjøberg, and Olav Egeland. Applied Runge-Kutta-Munthe-Kaas integration for the quaternion kinematics. *Journal of Guidance, Control, and Dynamics*, 42(12):2747–2754, 2019.

[90] Simon F. Tolić-Nørrelykke, Erik Schäffer, Jonathon Howard, Francesco S. Pavone, Frank Jülicher, and Henrik Flyvbjerg. Calibration of optical tweezers with positional detection in the back focal plane. *Review of Scientific Instruments*, 77(10), October 2006.

[91] Jana Tóthová and Vladimír Lisý. Overdamped and underdamped langevin equations in the interpretation of experiments and simulations. *European Journal of Physics*, 43(6):065103, October 2022.

[92] Firdaus E. Udwadia and Aaron D. Schutte. An alternative derivation of the quaternion equations of motion for rigid-body rotational dynamics. *Journal of Applied Mechanics*, 77(4), 04 2010. 044505.

[93] Firdaus E. Udwadia and Aaron D. Schutte. A unified approach to rigid body rotational dynamics and control. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 468(2138):395–414, 2012.

[94] S. Vadali. *On the Euler parameter constraint*, chapter 1, page 1. Aerospace Sciences Meetings. American Institute of Aeronautics and Astronautics, Jan 1988.

[95] Karen C. Vermeulen, Gijs J. L. Wuite, Ger J. M. Stienen, and Christoph F. Schmidt. Optical trap stiffness in the presence and absence of spherical aberrations. *Applied Optics*, 45(8):1812, March 2006.

[96] N. B. Viana, M. S. Rocha, O. N. Mesquita, A. Mazolli, P. A. Maia Neto, and H. M. Nussenzveig. Towards absolute calibration of optical tweezers. *Phys. Rev. E*, 75:021914, Feb 2007.

[97] Giorgio Volpe and Giovanni Volpe. Simulation of a brownian particle in an optical trap. *American Journal of Physics*, 81(3):224–230, March 2013. [doi:10.1119/1.4772632].

[98] R. A. Wehage and E. J. Haug. Generalized Coordinate Partitioning for Dimension Reduction in Analysis of Constrained Dynamic Systems. *Journal of Mechanical Design*, 104(1):247–255, 01 1982.

[99] Wikipedia. Gaussian beam - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Gaussian_beam. [Accessed 12-11-2023].

[100] Thomas Wriedt. Light scattering theories and computer codes. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 110(11):833–843, July 2009.

[101] Yuanjie Yang, Yu-Xuan Ren, Mingzhou Chen, Yoshihiko Arita, and Carmelo Rosales-Guzmán. Optical trapping with structured light: a review. *Advanced Photonics*, 3(03), may 2021.

[102] Wenjie Yu and Zhenkuan Pan. Dynamical equations of multibody systems on Lie groups. *Advances in Mechanical Engineering*, 7(3):1687814015575959, 2015.

[103] Mohammad Asif Zaman, Punnag Padhy, and Lambertus Hesselink. Fokker-planck analysis of optical near-field traps. *Scientific Reports*, 9(1):9557, Jul 2019.

[104] Nemanja D Zorić, Mihailo P Lazarević, and Aleksandar M Simonović. Multibody kinematics and dynamics in terms of quaternions: Langrange formulation in covariant form: Rodriguez approach. *FME Transactions*, 38(1):19–28, 2010.

BIOGRAPHICAL STATEMENT

Vatsal Joshi was born in Ahmedabad, Gujarat, India, in 1994. He received his B.E. degree from Gujarat Technological University, India, in 2015 and his M.S. degree from The University of Texas at Arlington in 2018 both in Mechanical Engineering. His areas of interest include computational dynamics, robotics, molecular dynamics and controls. Apart from these technical fields, he is also a hobbyist with huge interests in embedded systems and UAVs.