UNIVERSITY OF CALIFORNIA, MERCED

Stochastic Transport in Complex and Dynamic Geometries

A Dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in

Physics

by

Imtiaz Ahmad Ali

Committee in charge:

Professor Kevin Mitchell, Chair
Professor Chih-Chun Chien
Professor Arnold Kim
Professor Ajay Gopinathan

December 2021

The dissertation of Imtiaz Ahmad Ali, titled Stochastic Transport in Complex and Dynamic Geometries, is approved, and it is acceptable in quality and form for publication.

(Professor Ajay Gopinathan) Principal Adviser	Date
(Professor Kevin Mitchell) Committee Chair	Date
(Professor Chih-Chun Chien) Committee Member	Date
(Professor Arnold Kim) Committee Member	Date

University of California, Merced
2022

Dedication

This dissertation is dedicated to the memory of my grandparents, Sheikh Kamal Khan Maqbool, Badrul Nisha Maqbool, Amjad Ali and Fatima Bibi Ali.

Acknowledgments

I want to thank my advisor, Ajay Gopinathan. He is a great physicist and I have learned so much while working with him. I want to thank my committee members Kevin Mitchell, Arnold Kim and Chih-Chun Chien for their guidance and feedback on my work. I also want to thank the members of the Gopinathan group for all the scientific discussion and feedback. I would like to thank David Quint for all the teaching, support and intense discussion that really guided me through this journey. I want to thank my friends Mark De La Cruz Bartolo, Ahmad Elhares, Suryabhan Singh Hada, Bryan Maelfeyt, Carlos Adrian Morales, Anton Shvets, Simon Tekeste and Allan Wai for their support in my times of need. I want to thank my parents Nurun and Iftekhar and my cousins Fazleen Hanief and Stephanie Khan for their emotional support. Finally, I want to give a very special thanks to my wife Farnaz Golnaraghi for being a strong moral support. This research has been funded by the following sources:

- NRT Intelligent Adaptive Systems at UC Merced (NSF Grant No. DGE-1633722)
- NSF-CREST: Center for Cellular and Biomolecular Machines at UC Merced (NSF-HRD-1547848)

Imtiaz Ahmad Ali

136 North Spruce Avenue, South San Francisco, CA, 94080
Call: (650)-296-2317 , Email: imtiaza20@gmail.com

PROFILE

PhD candidate in Physics with research experience in Computational Biophysics and Statistical Mechanics. Experienced in computational modeling and analytical physics. Skilled in data analysis, data feature detection and data visualization. Experience in multiple programming languages/environments including, C++, Python and MATLAB. Have brought multiple coding projects from the ground up into the "it's working" stages with some experience in optimization. Excellent communication skills having presented my work at various national conferences. Worked in an interdisciplinary environment for the last 4 years. Passionate about computational work and enjoys being challenged with new types of problems. Fondness for learning on the fly and reworking solutions to problems until it is done correctly and efficiently.

EDUCATION

Master of Science in Financial Mathematics
The University of Chicago, August 2021 - December 2022 Expected

PhD Candidate in Physics
University of California, Merced, August 2016 - December 2021 Expected
Current research: Stochastic Transport in Complex and Dynamic Geometries
Principal Advisor: Professor Ajay Gopinathan

Master of Science in Physics
University of California, Merced, December 2019

Bachelor of Science in Physics with Minor in Mathematics
University of California, Santa Barbara, June 2016

COMPUTER SKILLS

Language & Software: C/C++, Python, R, MATLAB, Bash, LaTeX
Operating Systems: Linux, MS-Windows.

SELECTED COURSEWORK

Numerical Methods, Partial Differential Equations, Scientific Computing, Statistical Mechanics, Measurements & Uncertainties, Biophysics, Intelligent Adaptive Systems, C++ Programming, Molecular Dynamics, Machine Learning

SELECTED PROJECTS

- Intracellular Transport on Dynamic Actin Networks
  - Utilized C to build single-agent simulations to transport cargo on networks within a cell.
  - Developed theory to account for dynamical behaviors, and formulated dynamics of networks to make simulation closer to real world behaviors.
  - Utilized Python's object-oriented features and scientific libraries such as NumPy and SciPy to develop visual and modeling tools to quantify theorized assumptions.
- Lévy Walks in Curved Space

  - Built random walk simulation, in MATLAB, for transport on curved manifold by utilization of Lévy walk dynamics.
  - Detailed the mappings to different model spaces that made computations straightforward.
  - Developed analytical, visual and modeling tools to quantify dynamical behaviors.
- Optimal Models for Human Decision Dynamics
  - Gathered data from experiment that exposed subjects to high levels of stress through time constraints, limitations on available shelter space, and enforced group protocols that complicate the possibility of evacuation.
  - Developed human decision models, in MATLAB, which placed subjects in a natural disaster simulation; model addressed how individual decision behavior do not properly justify decision behavior will in groups.
  - Quantified human decision dynamics to find optimal human behavior for policy making in natural disasters, and to establish strategies that maximize the efficiency of human decision behaviors specifically in relation to minimizing threat and utilizing available information.

ACADEMIC SERVICE & PROFESSIONAL EXPERIENCE

- Teaching Assistant, Introductory Physics 2 Lab, Instructor: Dr. Carrie Manke, Spring 2021
- GRAD-EXCEL Peer Mentorship Program (AY 2018-2019, AY 2020-2021).
- Teaching Assistant, Modern Physics, Instructor: Dr. Kinjal Dasbiswas, Fall 2020
- Teaching Assistant, Introductory Physics 2, Instructor: Dr. Carrie Manke, Spring 2020
- Teaching Assistant, Calculus 1, Instructor: Matea Santiago, Fall 2019
- Teaching Assistant, Introductory Physics 2, Instructor: Kristina Callaghan, Fall 2018
- Teaching Assistant, Calculus 2, Instructor: Dr. Keith Thompson, Summer 2017
- Teaching Assistant, Calculus 1, Instructor: Daniel Swenson, Spring 2017
- Teaching Assistant, Calculus 2, Instructor: Haik Stepanian, Fall 2016

HONORS & AWARDS

- APS, Division of Biological Physics Student Travel Grant (Spring 2021)
- NSF-NRT IAS Fellow: University of California, Merced (Summer 2021)
- NSF-NRT IAS Fellow: University of California, Merced (Spring 2019 - Summer 2019)
- NSF-CREST CCBM Scholar: University of California, Merced (Fall 2018 - Present)
- NSF-NRT IAS Fellow: University of California, Merced (Fall 2017 - Summer 2018)
- NSF-NRT ICGE Fellow: University of California, Merced (Spring 2017)
- Academic Honors: University of California, Santa Barbara (2016)
- Distinction in the Major: University of California, Santa Barbara (2016)
- Research Honors Award: University of California, Santa Barbara (2016)
- Worster Fellowship: University of California, Santa Barbara (2015)

PUBLICATIONS

- **Imtiaz A. Ali**, Ajay Gopinathan, "Intracellular Transport on Dynamic Actin Networks", (In-Preparation).
- **Imtiaz A. Ali**, David A. Quint, Ajay Gopinathan, "Lévy Walks in Curved Space", (Submitted to Scientific Reports).
- K. J. Schlesinger, C Nguyen, **I. Ali** , J. M. Carlson, "Collective decision dynamics in group evacuation: Modeling Tradeoffs and Optimal Behavior." Xiv preprint arXiv:1611.09767 (2016).
- **I. Ali**, "Optimizing Models of Human Decision Making in a Natural Disaster". Undergraduate thesis, University of California, Santa Barbara, Santa Barbara, CA. 2016

CONFERENCE PRESENTATIONS

- Intracellular Transport on Dynamic Actin Networks, APS March Meeting, Spring 2021
- Levy Walks in Non-Euclidean Spaces, APS Far West, Fall 2018
- Random Searches in Non-Euclidean Spaces, APS March Meeting, Spring 2018
- Undergraduate Senior Physics Honors Thesis Defense, University of California, Santa Barbara, Committee Members: Dr. Jean M. Carlson, Dr. Mark Srednicki, Dr. Bjorn Birnir, May 2016
- Worster Fellow Presentation, University of California, Santa Barbara, Chair: Dr. Omer Blaes, Sep 2015

Abstract

Stochastic transport is a widely studied phenomenon among physicists. This includes simple diffusive processes like Brownian motion which have helped describe numerous systems ranging from the spreading of dye molecules in a liquid to the spreading of human populations. However, many natural processes exhibit anomalous diffusion with enhanced or suppressed spreading and can occur in environments that are complex. Transport behavior can be affected by properties such as the local curvature of a surface or the dynamics of a network on which the transport takes place. A quantitative characterization of these factors is critical for a deeper understanding of transport in such cases and is of much interest to the study of random walk theory, stochastic processes and anomalous diffusion in general. In this dissertation, we aim to accomplish this aim by focusing on two specific cases - (i) anomalous diffusion of a random walker on curved surfaces and (ii) transport of cargo on dynamic filament networks.

Lévy walks are a class of anomalously diffusive random walks with step lengths drawn from a heavy-tailed power- law distribution. They can be used to describe stochastic transport in a number of natural settings ranging from photons in complex media to animals foraging for food. Depending on the power-law exponent, Lévy walks in Euclidean spaces can show diffusive, super-diffusive or ballistic scaling of the mean-squared displacement. While such anomalous diffusive behavior has been extensively studied in Euclidean spaces, in many cases of interest the transport takes place on surfaces with non-zero Gaussian curvature, K, such as the membranes of cells or surfaces of planets. Here, we take the first steps towards studying how surface curvature affects anomalous transport described by Lévy walk statistics. We develop a computational model to simulate Lévy walks along geodesics in Euclidean ($K = 0$), spherical ($K > 1$) and hyperbolic ($K < 1$) spaces. By comparing our numerical results to a Taylor expansion of the mean-squared displacement (MSD) in powers

of curvature around the Euclidean case, we are able to establish the validity of a generalized expression for MSD of anomalous diffusion with curvature corrections and determine the expansion coefficients. Our results and methodology should be helpful in quantifying curvature contributions to processes such as reaction-diffusion and searches on curved surfaces as well as interpreting experimental data of anomalous diffusion on curved surfaces.

The transport of cargo within cells is a critical physiological process and is accomplished by a combination of simple diffusion and ballistic motor driven transport along cytoskeletal protein filaments giving rise to overall anomalous diffusive behavior. Recently there has been an explosion of new studies that consider the impact on transport of the morphology of the networks of filaments. This has been driven by advances in experimental techniques that have provided clearer and more controllable *in vivo* and *in vitro* studies. However, one aspect that has received much less attention is the growth/shrinkage and dynamic turnover of the network filaments themselves, which can occur on the same time scale as the transport of cargo on the network. Consequently, the complex intracellular dynamics of the inhomogeneous cytoskeletal structure can have profound impacts on transport. Here, we study transport of cargo carried by myosin motors on a dynamic actin network. We use a stochastic simulation model that accounts for both active cargo transport along filaments as well as passive diffusion and incorporate the dynamics of the explicitly represented actin network. We show how the speed of actin filament growth/shrinkage due to treadmilling affect cargo transport along with motor attachment/detachment rates and network density. We show the existence of filament dynamics in physiologically relevant regimes that optimize the transport of cargo and how it can be tuned by other system parameters. Our analysis illuminated how actin dynamics can be exploited by cells to modulate optimality in the molecular transport of cellular cargo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Overview

In nature, many phenomena evolve through stochastic transport, where randomness and fluctuations play a significant role. Stochastic transport can be described by the time evolution of the probability of an object being at a particular point in space. The simplest case is diffusion. Diffusion is the movement of particles from an area of higher concentration to an area of lower concentration driven by the random motion of individual particles. As an example, imagine being inside a closed room. If a bottle of perfume is sprayed, the scent particles would naturally diffuse from the spot where they left the bottle to all corners of the room. This diffusion would go on until the particles are equally distributed in the room. So, how do we characterize this diffusive spreading?

As particles move about the room, they undergo countless collisions with air molecules in the room. Their movements begin to resemble a random walk. Random walks can be done in different dimensions. For simplicity, let us consider a $1-$dimensional model starting at $x = 0$. The walker has a probability of stepping to the right and another to step to the left. The time evolution of the probability of particles is then described by the diffusion equation [4]. With enough steps, we get a probability distribution described by a Gaussian, which is a solution to the diffusion equation [4]. To understand the motion, we can look at how particles spread from their starting position as measured by the mean-squared displacement (MSD) which is given by, $\langle x^2 \rangle = s^2 = 2Dt^\alpha$, with exponent alpha equal to 1. This states that for a

simple diffusion process, the MSD scales linearly with time.

Not all processes are simple diffusion, nor do they occur in simple geometries. For example certain bacteria, swimming in sparse suspensions, exhibit a movement pattern called "run and tumble". Runs are straight trajectories [5]. Tumbles are shorter, random reorientation. This results in super-diffusive transport, $\alpha > 1$. Another example is turbulent fluid flow, which is a type of fluid flow where the fluid undergoes irregular fluctuations or mixing. The speed of the fluid at a point continuously changes in both magnitude and direction. It was shown, [6] that the diffusion process of particles in the turbulence can be approximated by a Lévy process. A Lévy process is a continuous time random walk, where the step lengths are drawn from a heavy-tailed distribution with Lévy exponent $\mu \in (2, 3)$. In this case also the MSD has a non-linear relation with time, $\alpha > 1$, such that the particles will exhibit super-diffusive motion.

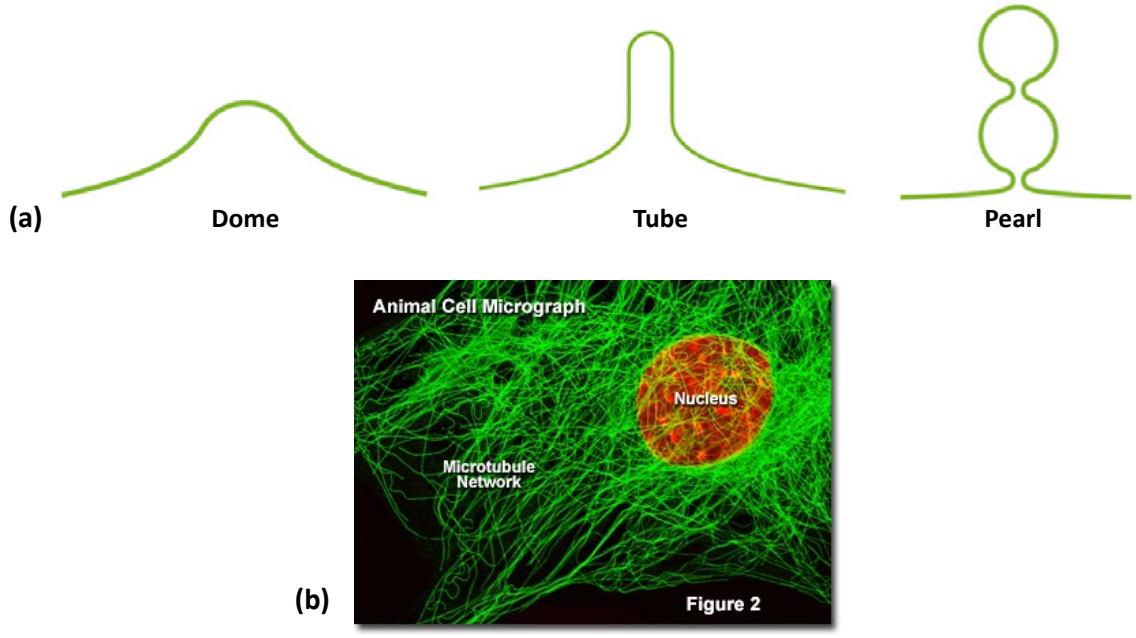

Figure 1.1: (a) Examples of complex membrane geometries that protein diffusion can occur on [1]. (b) A microtubule network inside an embryonic mouse cell with regions of crowded and sparse networks [2].

Systems described so far are for simple and anomalous diffusion but occurring in homogeneous, uniform flat space. However, many processes in nature occur in complex and dynamic geometries. For example cell membranes exhibit complex shapes

such as shallow domes, elongated tubular structures and pearl-like structures [1], shown in Figure 1.1a. The transport of proteins often occurs in membranes with such complex shapes, which can be the result of biological processes that require membrane deformation, such as endo- and exocytosis. The transport involves lateral diffusion of proteins, which are involved in signaling and sensing. We can ask, what is the effect of curvature on diffusion? Does it enhance or suppress diffusion? Another example is the transport of intracellular cargos which occurs in virtually all eukaryotic cells and is essential for many cellular functions [3]. Diffusion is sufficient for transport of small, nanometer (nm) scale molecules over small distances, less than microns (μm), like in bacteria. It becomes slow for larger cargo over large distances. These larger cargo require an active transport system where molecular motors carry the cargo along cytoskeletal filament network. In addition, the cytoskeletal network is dynamic, causing the networks geometry to continuously change with time. Changes in the network can cause regions to become crowded in some parts and in other parts they can be sparse, shown in Figure 1.1b. It is experimentally known that cargo transport is strongly influenced from the dynamics in network geometry [7]. We can ask, how do the dynamics affect cargo transport from one point to another in the cell? And, can the dynamics be tuned to optimize transport?

The results discussed here in this thesis address these questions in two parts. In chapter 2 we show how curvature affects anomalous diffusion. In chapter 3 we show how cargo transportation is affected when network of filaments become time-dependent. The common features connecting both of the presented projects are the tools of analysis. For transport on curved spaces, mean-squared displacement (MSD) is evaluated as a function of curvature. For intracellular transport, MSD and mean first-passage times (MFPTs) are evaluated as a function of the cytoskeletal dynamics and are used to characterize enhanced and optimal regions of transport. Much of what is written here, particularly in chapters two and three, are parts of papers that are either in preparation or have been submitted.

Chapter 2

Lévy Walks in Curved Space

2.1 Introduction

Random walks are stochastic processes that are used to describe statistical transport in a variety of systems ranging from the motion of molecules to the fluctuations of stock prices [4, 8–10]. A random walk describes a path resulting from a series of random steps in a space that are drawn from independent and identically distributed random variables. It could be thought of as a process by which randomly-moving objects wander away from where they started. Diffusion, which can be treated as the continuous limit of random walks, is the simplest and most fundamental transport mechanism in physical systems ranging from the interiors of cells to stars and is driven by random thermal motion [8]. Simple diffusion results in the mean squared displacement (MSD) of the transported entity increasing linearly with time [9]. In many cases, trapping or caging interactions with the environment or intermittent ballistic motion or anomalously large jumps can lead to the MSD being suppressed or enhanced resulting in sub-linear (sub-diffusion) or super-linear (super-diffusion) scaling of the MSD with time [11–13]. In other words, the MSD scales as $\langle \Delta s^2 \rangle \propto t^\alpha$, where α, the anomalous MSD exponent, could be greater than 1 (super-diffusive) or less than 1 (sub-diffusive). Examples of such anomalous transport include the sub-diffusion of passive particles within the cellular cytoplasm [14, 15] or confined environments [11] as well as the super-diffusion of actively transported material within cells [3, 16–18], the movement of swimming bacteria [9, 19] and particles in turbulent flows [12].

Lévy walks are a particular class of random walks that have been extensively used to describe super-diffusive transport in a variety of contexts ranging from photon transport in complex media to the foraging patterns of a variety of organisms [20–25]. Super-diffusive behavior in Lévy walks arises from the heavy-tailed power law nature of the probability distribution that the step lengths are drawn from ($P(l) \sim l^{-\mu}$). Depending on the Lévy exponent, μ, of the power-law, the scaling of the MSD with time can range from diffusive to ballistic [20], allowing for the description of a large variety of super-diffusive processes within the same framework. While anomalous diffusion, in general, and Lévy walks, in particular, have been extensively studied over the last few decades, our knowledge only extends to these processes occurring in flat or Euclidean spaces.

However, in many cases, these anomalous transport processes occur on surfaces with non-zero Gaussian curvature. For, example, the diffusion of membrane proteins on cellular membranes is typically sub-diffusive due to interactions with different membrane domains and the underlying cytoskeleton [26, 27]. But, these membranes are typically far from flat, exhibiting shapes with a range of positive and negative Gaussian curvatures [1] which are critical for a variety of cellular processes including migration, signaling and the intake of nutrients. At the other extreme, the super-diffusive transport of organisms, ranging from spider-monkeys and jackals [23, 24] to bacteria [25], occur on length scales that are not negligible compared to the naturally occurring radii of curvature present in their landscapes such as hills and valleys for animals, or the corrugations of the intestinal surfaces for bacteria. It is therefore critical to understand how the local geometry of the surface affects transport processes that occur on the surface. While there has been work on surface curvature contributions to simple or Brownian diffusion [1, 28–31], there have been no systematic studies of the impact of surface curvature on anomalous diffusion.

In this chapter, we take the first steps toward quantifying the dependence of super-diffusive Lévy walks on the surface curvature measured via corrections to the mean squared displacement (MSD). In section §2.2, we review the basic properties of Lévy walks and how the mean-squared displacement (MSD) depends on the Lévy exponent, μ, in flat space. We then derive a generalized series expansion for the MSD as a function of curvature by doing a Taylor expansion around the flat case. We then briefly summarize the random walk algorithm that we use to obtain numerical

results for MSD of Lévy walks on surfaces of arbitrary and constant curvature. In section §2.3, we present our numerical results and the fits to our generalized expansion for various values of μ on surfaces that are flat and with positive, and negative Gaussian curvatures. We show that the expansions work well to provide reliable estimates for the curvature corrections and provide estimates for the fit coefficients. Finally, in section §2.4, we discuss the implications of our results and the scope for future work.

2.2 Methods

In this section, we will go over geometry, curvature, Lévy distribution and mean-squared displacement along with the random walk algorithm.

2.2.1 Geometry and Curvature

Riemann geometry is the branch of differential geometry that studies Riemann manifolds, smooth manifolds with a Riemann metric, i.e., with an inner production the tangent space at each point that varies smoothly from point-to-point. This gives, in particular, local notions of angle, length of curves, surface area and volume. It deals with a broad range of geometries whose metric properties vary from point to point. An example of a curved geometry is the surface of a sphere and the two-sheeted hyperboloid, which is the focus of our work. In the context of this chapter, we define curved spaces as the space traversed on the surface of a given geometric object. Curved spaces can cause objects to become larger or smaller when compared to flat or Euclidean space. For instance, when comparing the area of a circle, from largest to smallest, hyperbolic has an exponential expression, Euclidean has a polynomial and spherical has a sinusoidal relation. Thus, hyperbolic space manages to pack in more surface area within a given radius than Euclidean. In contrast, spherical space will have less surface area compared to the Euclidean space. The reason is that the curvature of space directly affects the metric. Hyperbolic metric have negative curvature, causing objects, i.e., circle length or disk area, to grow exponentially. For Euclidean, the curvature is zero, flat space, causing a polynomial growth. With spherical space, the curvature is positive and induces a sinusoidal growth.

We can generalize our space to be non-Euclidean, meaning the metric does not have zero curvature. For example, visually, hyperbolic spaces can be thought of as smooth versions of trees abstracting the hierarchical organization of complex networks (non-trivial topological feature). Complex networks can be studied and described by embedding it to hyperbolic space [32]. It was shown that the curvature is directly related to network node degree distribution. The study of searches on scale-free networks, i.e., a network whose degree distribution follows a power-law [33], allows for the study of hyperbolic space, which is the natural embedding of them [32]. Some

examples of embedding scale-free networks to hyperbolic space are graphs [34], internet [35], and academic citations [36, 37]. Euclidean space is not capable of describing scale-free network space due to the way it grows (polynomial).

In addition, the presence of curvature is expected to affect the dynamics of a system. For example, in [38], they constructed a minimal model, to provide insight into the interplay between activity and geometry. They examined self-propelled particles subject to a realistic alignment rule, polar alignment, and white noise, confined to move on a sphere. They stated these systems are examples of motion from circulating band arising due to the incompatibility between spherical topology and uniform motion. They found that curvature indeed effects collective motion in active systems, leading to patterns not observed in Euclidean. Another example is flocks modeled by self-propelled particles that are confined to move on a sphere. All particles cannot travel at the same speed [38]. In this work, they concluded that frustration due to curvature leads to stable elastic distortions storing energy in the band. This allows for processes done on a spherical manifold to be studied. In addition, geographical systems, such as air navigation, utilize spherical space by modeling earth as a sphere [39].

2.2.2 Lévy Distribution and Mean-Squared Displacement

We examine the relationship of Lévy walks simulated in Euclidean ($K = 0$), spherical ($K > 1$) and hyperbolic ($K < 1$) space where K is the Gaussian curvature. We develop a simulation to perform random walks with respect to the spatial metric and show how the path taken has a dependence on curvature. Our work involves the investigation of anomalous diffusion in two-dimensions through Lévy walks implemented on spherical and hyperbolic manifolds in order to understand the structure and functionality of curvature.

Stochastic anomalous transport, in $2-$dimension, through Lévy walks have gained growing interest in the past two decades. Our study focuses on diffusive to superdiffusive regions of the mean-squared displacement, i.e., $\langle \Delta s^2 \rangle \propto t^{\alpha}$ with anomalous exponent $1 \leq \alpha \leq 2$. One aspect of interest is to see how random walk of Lévy processes within a curved metric affects anomalous transport. Our work involves performing stochastic diffusion process of a walker on different manifolds.

Lévy distribution fall into the category of a heavy-tailed distributions, a probability distribution whose tails are not exponentially bounded. Thus, the tails are heavier than an exponential distribution. The generalized Lévy distribution proof is omitted here but can be found in [40]. Here, we go over the process instead. By applying Fourier and Laplace transforms to the diffusion equation, $\frac{\partial \rho}{\partial t} = D\frac{\partial^2 \rho}{\partial \Delta s^2}$, the generalized probability density function, (ρ), has the following proportionality, $\rho(\Delta s) \propto (\Delta s)^{-\mu}$, with the Lévy exponent $\mu \in (2, 3)$ and the path length Δs. There are two main differences between Lévy flights and walks which need to be addressed. First, the flight assumes instantaneous transportation while Lévy walks have a physical speed. Second, due to the instantaneous transport, Lévy flights have divergent mean-squared displacement while Lévy walks are "finite". Lévy processes allow the searcher to cover more space of an area by searching locally then taking large jumps [12].

To implement a random walk with respect to geodesic length in our spaces, we need the correct probability relationship equations. Lévy distribution has two cases to consider when deriving it. First, when the upper bound on the distribution is finite, second when it is infinite. We show the final forms of the geodesic equations, and leave the proof in the Appendix §2.5.4.

For cases with restricted upper bounds, the geodesic path length is given as,

$$\Delta s(\Pr) = \left[\Delta s_{max}^{1-\mu} + \Pr\left(\Delta s_0^{1-\mu} - \Delta s_{max}^{1-\mu}\right)\right]^{\frac{1}{1-\mu}} \tag{2.1}$$

In cases with unrestricted upper bound, the geodesic path is,

$$\Delta s(\Pr) \approx \Delta s_0 \times (\Pr)^{\frac{1}{1-\mu}} \tag{2.2}$$

In the random walk simulation, our geodesic path length will be drawn from equation (2.1) only for spherical space and equation (2.2) for both Euclidean and hyperbolic with $\Pr(\Delta s > \Delta s_0) \in [0, 1]$. This allows us to traverse an unrestricted space. The sphere is bounded with a maximum geodesic length of $\Delta s = R \cdot \sigma_{central}$, with $\sigma_{central} \in (0, 2\pi)$ representing the central angle, such that we avoid overlapping the same geodesic path.

By using the probability to choose our geodesic length and not the density from equations (2.89, 2.92), we are able to avoid the instability of Lévy distributions. This was confirmed by generating a range of Δs from equations (2.1, 2.2), binning the

data and applying a fit for the parameters of equations (2.89, 2.92), respectively. The output was the respective density function with the corresponding μ, not shown.

Geometrical analysis is performed by computing the MSD. In section §2.1, we presented the general relation for MSD, $\langle \Delta s^2 \rangle \propto t^\alpha$. For Euclidean, in the limit of long time, the relation of the anomalous exponent α with the possible Lévy exponent, μ, is given by [13],

$$\langle \Delta s^2(t) \rangle \propto \begin{cases} t^2, & \text{for} \quad 1 < \mu < 2 \\ \frac{t^2}{\ln(t)}, & \text{for} \quad \mu = 2 \\ t^{4-\mu}, & \text{for} \quad 2 < \mu < 3 \\ t \cdot \ln(t), & \text{for} \quad \mu = 3 \\ t, & \text{for} \quad \mu > 3 \end{cases} \tag{2.3}$$

Ballistic motion is considered when $1 < \mu < 2$ with $(\alpha = 2)$, Lévy occurs for $2 < \mu < 3$ corresponding to $(\alpha = 4 - \mu)$ and Brownian for $\mu > 3$ with $(\alpha = 1)$. In analysis of Lévy random walks in Euclidean, a fit is performed, for the computed MSD from the simulated trajectories, for anomalous exponent $\alpha\,(\mu)$ with respect to these regimes. This ensures our random walk performs as stated in [13].

In Euclidean space, anomalous diffusion is described generally by $\langle \Delta s^2(t) \rangle_E = 4Dt^\alpha$. Consider the identical random walk in an isotropic space with constant Gaussian curvature $K_G = \pm \frac{1}{R^2}$ where R is the radius of curvature. The MSD in such a curved space must be $\langle \Delta s^2(t) \rangle_{N.E} = f\,(D, t, \alpha, K_G)$. Now consider the curvature as a perturbation from flatness, $K_G = 0$. By Taylor expanding around $K_G = 0$ gives,

$$\begin{aligned} \langle \Delta s^2(t) \rangle_{N.E} =& f\,(D, t, \alpha, K_G)\Big|_{K_G=0} + \\ & K_G \frac{\partial f\,(D, t, \alpha, K_G)}{\partial K_G}\Big|_{K_G=0} + \frac{K_G^2}{2} \frac{\partial^2 f\,(D, t, \alpha, K_G)}{\partial K_G^2}\Big|_{K_G=0} + \cdots \end{aligned} \tag{2.4}$$

Now $f\,(D, t, \alpha, K_G = 0) = 4Dt^\alpha$ and set $\frac{\partial f}{\partial K_G}|_{K_G=0} = g_1\,(D, t, \alpha)$. Note, from equation (2.4), we see the term $K_G \cdot g_1\,(D, t, \alpha)$ has dimensions of $\left[L^2\right]$, hence g_1 has dimensions of $\left[L^4\right]$. The only length scale one can construct out of D, t, α is $(Dt^\alpha)^{\frac{1}{2}}$, therefore $g_1\,(D, t, \alpha) = \gamma_1\,(Dt^\alpha)^2$ where γ_1 is a dimensionless constant that could

depend on α. Similarly, $g_2(D,t,\alpha) = \gamma_2 (Dt^\alpha)^3$. Plugging this into equation (2.4) gives,

$$\begin{aligned}\langle \Delta s^2(t)\rangle_{N.E} &= 4Dt^\alpha + \gamma_1 \frac{(Dt^\alpha)^2}{R^2} + \frac{\gamma_1}{2}\frac{(Dt^\alpha)^3}{R^4} + \cdots \\ &= 4Dt^\alpha \cdot \left[1 + \widetilde{\gamma_1}\left(\frac{Dt^\alpha}{R^2}\right) + \widetilde{\gamma_2}\left(\frac{Dt^\alpha}{R^2}\right)^2 + \cdots\right]\end{aligned} \tag{2.5}$$

This gives the MSD for curved spaces in terms of that space but multiplied by a dimensionless constant that depends on geometry,

$$\langle \Delta s^2(t)\rangle_{N.E} = \langle \Delta s^2(t)\rangle_E \cdot \left[1 + \widetilde{\gamma_1}\left(\frac{Dt^\alpha}{R^2}\right) + \widetilde{\gamma_2}\left(\frac{Dt^\alpha}{R^2}\right)^2 + \cdots\right] \tag{2.6}$$

This expression is valid in the regime that $\frac{Dt^\alpha}{R^2} \leq 1$, i.e., when the MSD in Euclidean space is small compared to the inverse Gaussian curvature.

With Euclidean random walk, we implement the same methodology for the hyperbolic and spherical space Lévy walk, but with its respective position and geodesic equations, derived in the Appendix (§2.5.2, §2.5.3). Additionally, because of the scale-free behavior of Lévy distributions, considering only Brownian motion and following the procedure in [41], the generalized expression for the MSD in curved space is,

$$\langle \Delta s^2(t)\rangle \approx 2dDt - \frac{2}{3}R_g (Dt)^2 + \frac{4}{45}\frac{d-3}{d(d-1)}R_g^2 (Dt)^3 + \cdots \tag{2.7}$$

where d is the number of dimensions, D is the diffusion coefficient, and $R_g = \pm\frac{d(d-1)}{R^2}$ is the scalar curvature. $R_g > 0$ for positive curvature, $R_g < 0$ for negative and $R_g = 0$ for zero curvature. In terms of equation (2.6), with $\langle \Delta s^2(t)\rangle_E = 2dDt$ and $\alpha = 1$, we get $\widetilde{\gamma_1} = \pm\frac{1}{3}\cdot(1-d)$ where $(+)$ is for $R_g > 0$ and $(-)$ for $R_g < 0$ and $\widetilde{\gamma_2} = \frac{2}{45}\cdot(d-3)\cdot(d-1)$.

In deriving equation (2.7), they used the generalized Laplace-Beltrami operator instead of Euclidean Laplace operator in the diffusion equation [41]. This properly accounts for the curvature effects which takes place for motion on a Riemann manifold. The derivation was justified by working in the localized frame of Riemann Normal Coordinates (RNC).

For consistency, with the proposed Brownian motion MSD, we adopt the same

sign orientation from equation (2.7) for the dimensionless constants, $\widetilde{\gamma_1}, \widetilde{\gamma_2}$, in equation (2.6). The generalized form of anomalous diffusion by Lévy random walk in d-dimension, with $d = 2$ and $R_g = \pm\frac{2}{R^2}$, where $(+)$ is for spherical and $(-)$ is for hyperbolic space is given by,

$$\langle \Delta s^2(t) \rangle \approx \begin{cases} 4Dt^2 \cdot \left[1 \pm \widetilde{\gamma_1}\left(\frac{Dt^2}{R^2}\right) - \widetilde{\gamma_2}\left(\frac{Dt^2}{R^2}\right)^2 + \cdots\right], & \text{for} \quad 1 < \mu < 2 \\ 4Dt^{4-\mu} \cdot \left[1 \pm \widetilde{\gamma_1}\left(\frac{Dt^{4-\mu}}{R^2}\right) - \widetilde{\gamma_2}\left(\frac{Dt^{4-\mu}}{R^2}\right)^2 + \cdots\right], & \text{for} \quad 2 < \mu < 3 \\ 4Dt \cdot \left[1 \pm \frac{1}{3}\left(\frac{Dt}{R^2}\right) - \frac{2}{45}\left(\frac{Dt}{R^2}\right)^2 + \cdots\right], & \text{for} \quad \mu > 3 \end{cases} \tag{2.8}$$

Our method expresses the expansion with respect to time, since in Euclidean, the MSD equations (2.3) only affects time and not the diffusion constant.

2.2.3 Random Walk Algorithm

Implementing a random walk requires the walker to remain on the surface of our geometrical object. By introducing curvature, the position equations change, but the methodology of advancing the walk remains unchanged. The equations are shown, for each space, in the Appendix §2.5.4. Here, we state what the geodesics are and the process of implementing them for each space. In Euclidean, geodesic paths are straight lines with position equations (2.95). In choosing the Euclidean geodesic path length, Δs, we break the path into n fixed step lengths, δ, where $\Delta s = \sum^n \delta$.

For hyperbolic space, using the half-plane model, geodesics are semi-circles. In order to implement a random walk in this space, we formulated a local approximation to pick, randomly, the semi-circle center, show in the Appendix §2.5.2. Semi-circles are produced by using the position equations (2.96). For hyperbolic geodesic path length, Δs, each step is the sum of everything prior for the geodesic, such that $\Delta s = \sum_1^m \delta$, where $m \in [1, n]$. Both Euclidean and hyperbolic use geodesic equation (2.2) with polar angles drawn form a uniform distribution, $\theta = 2\pi \cdot \Pr$.

In spherical space, geodesics are arcs. These arcs are produced by the position equations (2.97), achieved by using the method of quaternions, discussed in the Appendix §2.5.3. Spherical space uses geodesic equation (2.1), where the geodesic path

length, Δs, is broken into n fixed step lengths, δ, where $\Delta s = \sum^{n} \delta$. This gives the incremental central angle, $\sigma_c = \frac{\delta}{R}$, used as the amount rotated about a randomly chosen unit vector. As the walker traverses the path, our unit vector remains unchanged.

Our walker starts with randomly picked initial positions, which for Euclidean and hyperbolic are, (x_0, y_0), and for spherical, (x_0, y_0, z_0). Starting the walk, geodesic distance, Δs, is chosen at random from the corresponding Lévy relation associated for that space. For simplicity, we floor the value of Δs to the nearest integer, so we have discrete walks. By using the iterative method, stated in the Appendix §2.5.4, we are able to traverse the full geodesic path. The new coordinates are updated by equation (2.95, 2.96, 2.97), respective to each space. A sample of our simulation, for hyperbolic and sphere space, is shown in Figure 2.1. The simulations and analyses are done using MATLAB. Our simulation has a stepping length of $\delta = 1$ and speed $v = 1$. For all three geometries, the walk is broken up into fixed step lengths, with constant speed v, such that the total time to traverse the geodesic path Δs is $t = \frac{\Delta s}{v}$.

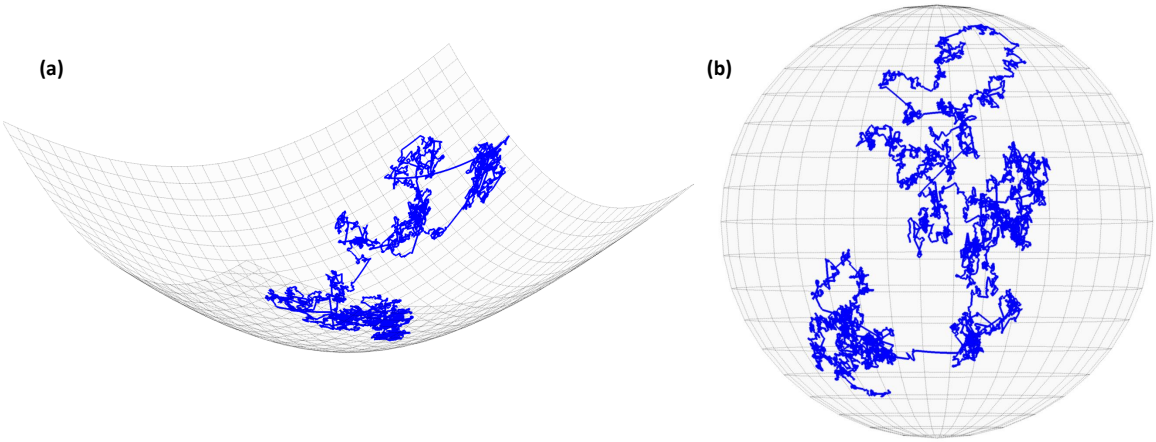

Figure 2.1: Simulated random walk on geodesics with Lévy exponent $\mu = 3.4$ with $R = 100$ for $t = 10,000$ time-steps. (a) Random walk trajectory on the surface of hyperboloid with negative Gaussian curvature according to the metric in the Appendix §2.19. (b) Random walk trajectory is on the surface of a sphere corresponding to positive curvature.

2.3 Results

We start by going over the various fit models, preliminary Euclidean MSD data, Brownian motion for spherical and small curvature hyperbolic, then Lévy and ballistic

motion only for the sphere.

2.3.1 Model and Fits

Our analysis consists of modeling five MSD forms for 2−dimensional random walk, shown in equation (2.9). In this form, it is computationally beneficial to multiply through equation (2.8) while keeping the expansion term in parenthesis dimensionless such that, for Lévy and ballistic cases, $\widetilde{\beta}_1 = 4 \cdot \widetilde{\gamma}_1 \cdot R^2$ and $\widetilde{\beta}_2 = 4 \cdot \widetilde{\gamma}_2 \cdot R^2$; whereas for Brownian, $\widetilde{\beta}_1 = \frac{4}{3} \cdot R^2$ and $\widetilde{\beta}_2 = \frac{8}{45} \cdot R^2$. This also allows us to look at the expansion terms in first, second and third order. The notations of the MSD model is, $\langle \Delta s^2(t) \rangle_{i,j}$, where i represents the polynomial order and j as the number of parameters being fitted. The negative sign, $(-)$, on the second order term corresponds to spherical space and positive, $(+)$, for hyperbolic. The additional terms are explained in section §2.2.2. From visual inspection, the additional terms will either have positive or negative effects on our random walk. Therefore, we expect spherical space to cover less distance and for hyperbolic to cover more when compared to Euclidean.

$$\langle \Delta s^2(t) \rangle_{1,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \tag{2.9a}$$

$$\langle \Delta s^2(t) \rangle_{2,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \frac{4}{3}R^2 \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^2 \tag{2.9b}$$

$$\langle \Delta s^2(t) \rangle_{3,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \frac{4}{3}R^2 \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^2 - \frac{8}{45}R^2 \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^3 \tag{2.9c}$$

$$\langle \Delta s^2(t) \rangle_{2,3} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \widetilde{\beta}_1 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^2 \tag{2.9d}$$

$$\langle \Delta s^2(t) \rangle_{2,3} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \widetilde{\beta}_1 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^2 - \widetilde{\beta}_2 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2} \right)^3 \tag{2.9e}$$

The anomalous exponent a will always have the range shown in equation (2.3), depending on the Lévy exponent μ, and is considered a known, fixed value in the fit. Greek letters, $\widetilde{\alpha}, \widetilde{\beta}_1$, along with the diffusion constant $\widetilde{D}$, are parameters that will be fitted. In this work, we only present results of the best-fit with the least number of terms and parameters. Thus, results for equations (2.9c, 2.9e) are excluded from this presentation because the third order term has very little contribution for prediction within our time regime. In addition, our work consists of fitting a total of

eight models, an additional three models to the models shown above. The full list can be found in the Appendix §2.5.4. The fit will be done in MATLAB using nonlinear least-squares method.

To validate our fit, we compute the root-mean-squared error (RMSE) and the adjusted R-squared. Due to our MSD being an expansion, we require that our prediction is up to a maximum time range such that the MSD does not exceed 10% of R^2. The range of fit, to predict our accepted range, is determined by the lowest value of our RMSE and corresponding adjusted R-squared. For example, consider the sphere with radius $R = 10$, simulation Lévy exponent of $\mu = 3.4$ and using equation (2.9b) as the model we want to fit. In addition, we fit equation (2.9a) to further show the need for our expansion. The lowest RMSE value occurs using equation (2.9b) to fit 73% of the data, 55 time-steps, with $RMSE = 4.02$ and corresponding adjusted R-squared of 0.42. In the case of using equation (2.9a), the $RMSE = 5.1$ with adjusted R-squared of 0.08. Therefore, by using equation (2.9b) to fit 55 time-steps, we are able to most accurately predict up to 75 time-steps. Interestingly, even though both equations used for the fit are penalized for 2 parameters, the expansion term version gives the best fit. The results can be found in the supplement. This allows us to exploit the differences between random walk done on different manifolds with curvature versus zero curvature (Euclidean). After finding the appropriate ranges that best models our results, we present the MSD plots comparing the models in equation (2.9) to the simulated MSD results. From here, the same method will be used to find our fitting range, therefore we will omit showing the RMSE and adjusted R-squared data. The fitting and maximum prediction range are presented in the supplement along with each models fitted parameter values. The results presented, uses the fitted parameters for the specified models unless stated otherwise.

2.3.2 Results: Preliminary Euclidean Case

Before presenting the curved space work, we test our methodology in Euclidean space to insure the consistency in our approach, therefore reproducing the results given by equation (2.3). Our work excludes Lévy exponents $\mu = 2, 3$. Shown in Figure 2.2 are the MSD results for random walks in Euclidean space. The geodesics of simulated trajectories are drawn from equation (2.2), which have Lévy exponent $\mu = 1.6, 2.4, 2.8, 3.4$ with the corresponding theoretical anomalous exponent

$\alpha = 2, 1.6, 1.2, 1$, respectively. The simulations are for 100 samples, each with 1000 time-steps. By applying equation (2.98) to the simulated trajectories and taking the ensemble-average of the time-averaged results, we get the simulations MSD, represented by red dashes and gray regions as the 95% confidence interval. To confirm our random walk method, a fit is performed on the simulations MSD by equation (2.9a), represented by green lines.

Starting with ballistic motion, the simulation is run with Lévy exponent $\mu = 1.6$, with its calculated MSD shown in Figure 2.2a. We predict up to 25 time-steps in which 60% of the data is used for the fit, green. The fit has anomalous exponent of $\widetilde{\alpha} = 1.955 \pm 0.002$ where percentage error is 2.3% from theory. Indeed, our random walk simulation reproduces ballistic behavior. In the case of Lévy motion with $\mu = 2.4$, where its calculated MSD is shown in Figure 2.2b. The prediction is for 250 time-steps with 60% of the data used for the fit. The fit produces the anomalous exponent of $\widetilde{\alpha} = 1.561 \pm 0.003$ with percentage error of 2.4% from theory. Our simulation produces the expected Lévy like behavior, where the anomalous exponent scales as $\alpha = 4 - \mu$. For Brownian, the simulated trajectories are produced with Lévy exponent of $\mu = 3.4$, Figure 2.2c. The expected anomalous exponent is $\alpha = 1$ such that MSD scales linearly with time. For predicting 350 time-steps and fitting 71% of the data, the outputted anomalous exponent is $\widetilde{\alpha} = 1.036 \pm 0.000$. This has a percentage error of 3.6% from theory. From the plot, we see a linear relationship between MSD and time. Therefore, Brownian-like behavior occurs for our simulation.

Overall, the methodology to produce the stated motions in our simulation remains robust. Our simulation reproduces the theorized Euclidean results with our fitted MSD anomalous exponent $\widetilde{\alpha}$ within 5% error, shown in Figure 2.2d with plots of the actual $\widetilde{\alpha}$ versus μ. Given these results, the methodology used to simulate random walks in Euclidean space will also be used to produce trajectories along geodesics for spherical and hyperbolic space with its respective dynamical equations.

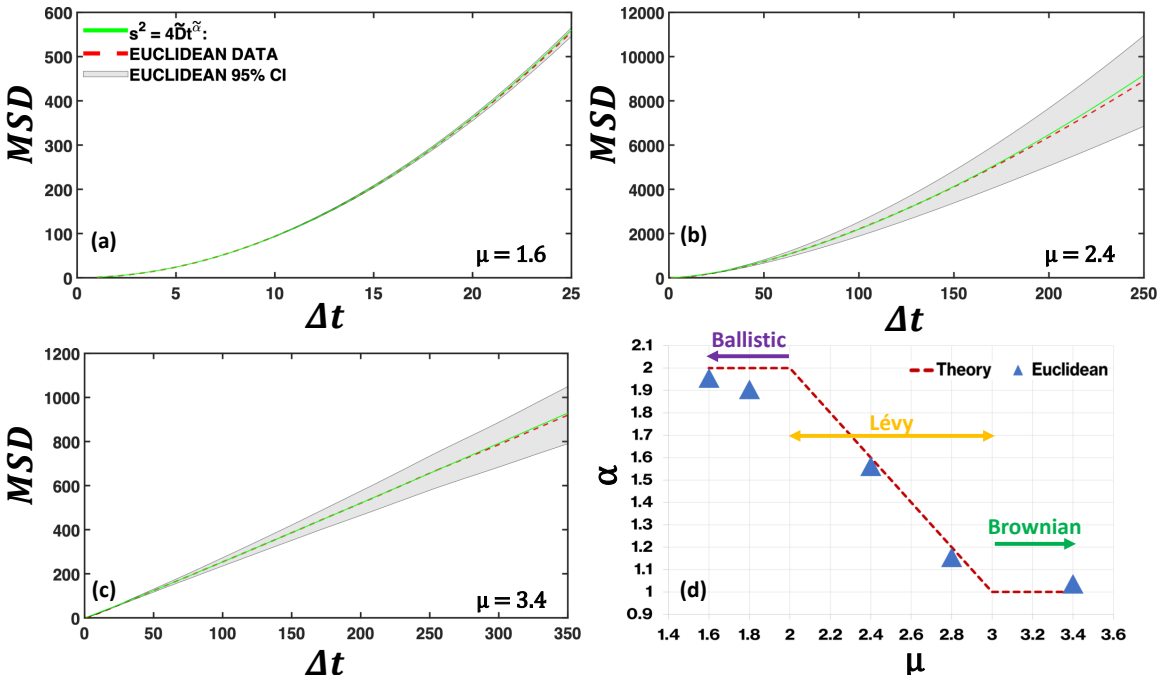

Figure 2.2: Shown are the MSD results for random walks in Euclidean space. The simulated trajectories geodesics are drawn from equation (2.2), which have Lévy exponent $\mu = 1.6, 2.4, 2.8, 3.4$ with the corresponding theoretical anomalous exponent $\alpha = 2, 1.6, 1.2, 1$, respectively. The simulation consists of 100 samples in which the computed MSD is represented by red dashes and gray regions as the 95% confidence interval. A fit is performed on the simulations MSD by equation (2.9a) and plotted with green lines. (a) $\mu = 1.6$, we have ballistic motion. We predict up to 25 time-steps in which 60% of the data is used for the fit, green. The fit has anomalous exponent of $\widetilde{\alpha} = 1.955 \pm 0.002$ which is within 2.3% from the theorized value and a diffusion constant of $D = 0.259 \pm 0.002$. (b) $\mu = 2.4$, we have Lévy motion. The prediction is for 250 time-steps in which 60% of the data is used for the fit. Our simulation shows an exponent of $\widetilde{\alpha} = 1.561 \pm 0.003$ which is within 2.4%, diffusion constant of $D = 0.414 \pm 0.006$. (c) $\mu = 3.4$, we have Brownian motion. The prediction is for 350 time-steps in which 71% of the data is used for the fit. Our simulation shows an exponent of $\widetilde{\alpha} = 1.036 \pm 0.000$ which is within 3.6%, diffusion constant of $D = 0.537 \pm 0.001$. For Brownian, the expected anomalous exponent is $\alpha = 1$ such that MSD scales linearly with time. (d) Plots of the actual $\widetilde{\alpha}$ vs μ. Our methodology reproduces the theorized Euclidean results with our fitted MSD anomalous exponent $\widetilde{\alpha}$ within 5% error and remains robust.

2.3.3 Results: Brownian Case

In the previous section §2.3.2, we showed the robustness of our methodology by simulating random walks in Euclidean space, reproducing the expected anomalous

diffusive relations, equation (2.3). In this section, we move over to curves space and present the simulated random walk MSD results for spherical and hyperbolic space by utilizing the same methodology used for Euclidean, specifically for Brownian motion. For spherical space, the simulated results are produced for 100 samples with $R = 10, 100$ and for a maximum of $500, 1000$ time-steps, respectively. For hyperbolic, we use the half-plane model, with $R = 100, 1000$ for 25 samples and ran for 2000 time-steps.

The curved space MSD is calculated from the simulated trajectory data using equation (2.99) for the sphere and equation (2.100) for hyperboloid, shown in red dashed line, with its 95% confidence interval as the gray shaded regions. To compare, the Euclidean results are shown as black dashed line with the yellow region as its 95% confidence interval, allowing better understanding of the correction terms in equation (2.8) and the impact of curvature. In our analysis and focus on Brownian motion, the computed MSD from the simulated trajectories will be fitted by using equation (2.9b), shown in magenta. In order for the fit to work, we require our prediction MSD range to not surpass 10% of R^2. From reference [32], we expect the simulations calculated MSD, for the Euclidean data, to remain above spherical and below hyperbolic space.

Results: Sphere, Brownian, $\mu = 3.4$

To understand the curvature influence in diffusive processes, we start by simulating Brownian motion in spherical space for $R = 10, 100$. The simulated trajectories are produced with Lévy exponent of $\mu = 3.4$ for 100 samples, shown in Figure 2.3. Using equation (2.99), we compute the MSD for the sphere, plotted with red-dashed line, with its 95% confidence interval as the gray shaded regions. Euclidean results are shown with black dashed line with the yellow region as its 95% confidence interval. This spherical space MSD is fitted to equation (2.9b) shown in magenta. The expected anomalous exponent is $\alpha = 1$ such that MSD scales linearly with time, which holds for Euclidean. We expect the second order term to cause larger deviation from Euclidean with greater curvature and increasing time without deviating from the expected anomalous exponent α.

For spherical space with radius $R = 10$, the resulting MSD is shown in Figure 2.3a. To remain in our threshold, 10% of R^2, the prediction is for 75 time-steps in which 73%

of the data is used for the fit, magenta. The fit produces an anomalous MSD exponent of $\widetilde{\alpha} = 1.013 \pm 0.017$ with an error of 1.3% from theory. The corresponding diffusion constant is $\widetilde{D} = 0.329 \pm 0.021$. From visual inspection, we see Euclidean data (black-dashes) above spherical with increasing deviation. To better understand the spread, by utilizing the fitted parameters, we look at the percentage difference of the second and third order term to the first order term of equation (2.9c). The second order term has a percentage difference of $5.8\%, 8.7\%$ for time-steps $t = 50, 75$, respectively; whereas the third order term percentage difference is $0.15\%, 0.29\%$. Clearly, the second order term dominates over the third term. In addition, the second term has an increasing contribution.

With spherical radius of $R = 100$, we predict up to 400 time-steps where 75% of the data is used for the fit, shown in Figure 2.3b. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 0.979 \pm 0.008$ with percentage error from theory of 2.1%. The corresponding diffusion constant is $\widetilde{D} = 0.551 \pm 0.023$. Visually, Euclidean data still remains above spherical with increasing deviation, but not as significant compared to $R = 10$. Similarly for $R = 10$, for time-steps $t = 200, 400$, the second term percentage differences are $0.33\%, 0.64\%$ and the third term with $0.0005\%, 0.001\%$. The second term dominates over the third term and has an increasing contribution.

In both cases, the Euclidean MSD remains above the spherical. This shows diffusive motion on the sphere is negatively impacted from curvature when compared to Euclidean. At some point, if considering only the first order term, we would expect the fit to follow the Euclidean data. As time progresses, adjustment is made by the second order term, signaling the need of an additional term. This point can be thought of as the minimum time needed to see curvature effects on the random walk. Moreover, the third order term has very little influence and can be neglected. In addition, as we increase R, decreasing the curvature, we see the spread from Euclidean and spherical space decrease, indicated by the second term percentage contribution from $R = 10$ and $R = 100$.

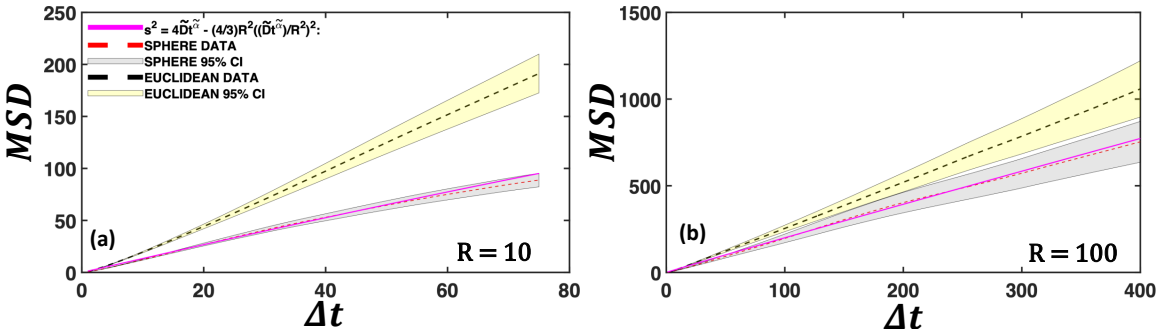

Figure 2.3: Shown are the MSD results for random walks in spherical space for $R = 10$ and 100. Geodesic step-lengths are drawn from equation (2.1), with Lévy exponent $\mu = 3.4$ to produce Brownian motion. The theoretical anomalous exponent is $\alpha = 1$. The simulation consists of 100 samples and the computed MSD are the red-dashed line and gray regions as the 95% confidence interval. A fit is performed on the simulation MSD, (red-dashes), by equation (2.9b), shown by the magenta lines. Euclidean results are the black-dashed line with the yellow region as its 95% confidence interval. (a) $R = 10$. We predict up to 75 time-steps in which 73% of the data is used for the fit. The fit has anomalous exponent of $\widetilde{\alpha} = 1.013 \pm 0.017$, which is within 1.3% from theory and a diffusion constant of $\widetilde{D} = 0.329 \pm 0.021$. (b) $R = 100$. We predict up to 400 time-steps with 75% of the data used for the fit. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 0.979 \pm 0.008$, which is within 2.1% error, and a diffusion constant of $\widetilde{D} = 0.551 \pm 0.023$. In both cases, Brownian motion in Euclidean remains above spherical and the deviation increases with later time. For $R = 10$, larger curvature, the spread is more significant.

Results: Hyperbolic, Small Curvature, Brownian, $\mu = 3.4$

In the case of Brownian motion for spherical space, previous section §2.3.3, we were able to show how curvature affects diffusive processes. As curvature decreases, spherical results start to look more like Euclidean. In this section, we focus on hyperbolic space for small curvature. Due to the exponential growth of this space, we exclude analyzing the case of large curvature, i.e., to remain in our threshold of 10% of R^2 we do not look at $R = 1, 10$, because insufficient number of data points are available to perform a fit. Our objective is to show where the curvature influence on motion is equivalent to Euclidean space. We have our hyperbolic space with $R = 100, 1000$. The simulated trajectories are produced for Brownian motion with Lévy exponent of $\mu = 3.4$ for 25 samples each with 2000 time-steps, shown in Figure 2.4. The MSD is computed using equation (2.100), plotted with red-dashed line, with its 95% confidence interval as the gray shaded regions. The data is fitted by equation (2.9b),

shown in magenta. To emphasize the effects of small curvature, Euclidean results are included represented by black-dashed line with the yellow region as its 95% confidence interval. We expect the anomalous exponent to be around 1, $\alpha = 1$, such that MSD scales linearly with time. Further, we expect the second order correction term to have minimal to no influence on the diffusive process, for our time range.

When considering small curvature, our assumed threshold of 10% of R^2 need not hold. Instead, a new condition is made such that the expansion terms contribution is small. For our time range to work for small curvature, we require the following condition to hold for the second order term of our expansion, $\frac{4}{3}R^3\left(\frac{Dt^\alpha}{R^2}\right)^2 \leq \epsilon$, where ϵ is the largest allowed MSD correction to the first order term. For example, if the first order term has a value of 100, allowing 10% yields $\epsilon = 10$. Solving for time, our conditional requirement is $t \leq \left(\frac{R}{D}\left(\frac{3}{4}\cdot\epsilon\right)^{\frac{1}{2}}\right)^{\frac{1}{\alpha}}$, thus, $t \propto R^{\frac{1}{\alpha}}$. For completeness, applying the same method, the condition for our third order term is, $t \leq \left(\frac{R^{\frac{4}{3}}}{D}\left(\frac{45}{8}\cdot\epsilon\right)^{\frac{1}{3}}\right)^{\frac{1}{\alpha}}$, therefore $t \propto R^{\frac{4}{3\cdot\alpha}}$. The results presented below have parameter values fitted for less than the prediction range.

For hyperbolic space with $R = 100$, the resulting MSD is shown in Figure 2.4a, the prediction is for 800 time-steps in which 38% of the data is used for the fit, magenta. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 1.043 \pm 0.002$ which has an error of 4.3% from theory and diffusion constant of $\widetilde{D} = 0.537 \pm 0.006$. Visually, Euclidean MSD (black-dashes) aligns with hyperbolic. Similar to the spherical case for Brownian motion, for time-steps $t = 400, 800$, the percentage difference of the second order term is $0.93\%, 1.9\%$, respectively, and the third term is $0.004\%, 0.01\%$. Even though the second term dominates over the third, both have very little contribution. Further, using our small curvature time range condition, the second terms fitting range is up to 802 time-steps, with the third term 797. The fitted result are for 38% of 800 or 304 time-steps. Therefore, our range of fit is well within our conditions.

With $R = 1000$, we predict up to 800 time-steps where 63% is used for the fit, shown in Figure 2.4b. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 1.014 \pm 0.004$, which is 1.4% from theory, and a diffusion constant of $\widetilde{D} = 0.709 \pm 0.017$. Visually, Euclidean still aligns with hyperbolic. Similar to hyperbolic case with $R = 100$, the percentage difference for the second term is $0.01\%, 0.02\%$ for time-steps $t = 400, 800$, respectively; whereas for the third order terms is approximately zero. Indicating that

the additional curvature terms have no significant impact on the diffusive process. Using our small curvature time range condition, the second terms fitting range is up to 787 time-steps and 782 for the third order term. The fitted result are for 63% of 800 corresponding to 504 time-steps, which is well within our condition.

The computed MSD from the simulation data, red-dashes for hyperbolic and black-dashes for Euclidean, have similar behavior with both curves aligning within the 95% confidence interval, gray and yellow shaded regions. The coefficient of the second order term, $\frac{4}{3\cdot R^2}$, is small enough causing no significant influence. In both cases, the third terms contributions are negligible and can be excluded. Moreover, our small curvature time range conditions hold, thus, for small principal curvature, $(\frac{1}{R})$, hyperbolic space results become equivalent to Euclidean.

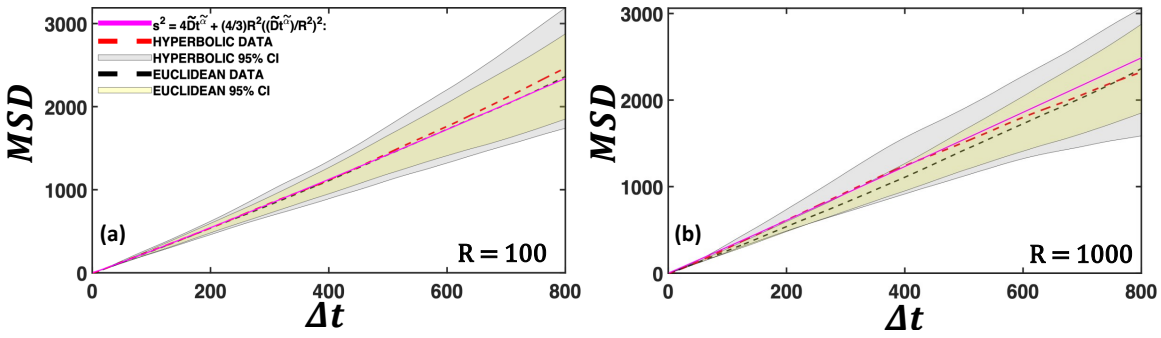

Figure 2.4: Shown are the MSD results of random walks, for small curvature, in hyperbolic space with $R = 100$ and 1000. Geodesic step-lengths are drawn from equation (2.2), with Lévy exponent $\mu = 3.4$ to produce Brownian motion. The simulation consists of 25 samples in which the computed MSD is represented by red dashes and gray regions as the 95% confidence interval. A fit is performed on the simulations MSD, (red-dashes), by equation (2.9b), shown by the magenta lines. Euclidean results are the black-dashed line with the yellow region as its 95% confidence interval. (a) $R = 100$. We predict up to 800 time-steps in which 38% of the data is used for the fit. The fit has anomalous exponent of $\widetilde{\alpha} = 1.043 \pm 0.002$, which is within 4.3% from theory and a diffusion constant of $\widetilde{D} = 0.537 \pm 0.006$. (b) $R = 1000$. We predict up to 800 time-steps with 63% of the data used for the fit. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 1.014 \pm 0.004$, within 1.4% error, and a diffusion constant of $\widetilde{D} = 0.709 \pm 0.017$. The second term of our expansion contributes less than 2% when compared to the first term, as explained in the main text. Therefore, with small principal curvature, $(\frac{1}{R})$, hyperbolic space behaves equivalently to Euclidean.

Overall, our results align with our prediction for Brownian motion, equation (2.8), within the assumed time regime where the simulation MSD is 10% of R^2. Random walks performed in curved spaces affect the motion by intrinsic properties that need to

be accounted for. Without having the curvature correction term, we would be misled to think the behavior changes, for example Lévy cases will turn into ballistic or vice versa. From the presented results, only the second order term is required to justify our work, therefore we can exclude using the third order term in equation (2.9c) and instead work with equation (2.9b). By performing that same random walk methodology for Euclidean, we are able to reproduce the same behavior for spherical and hyperbolic space by accounting for curvature corrections for long enough time regimes, where the MSD is within 10% of R^2.

In the limit of small curvature, where we see hyperbolic results align with Euclidean, Figure 2.4, we speculate the same results for the sphere, a shared limitation for both spaces. The simulation results indicate that diffusion on a sphere negatively impacts MSD, and by the same logic, we speculate hyperbolic space to have a positive impact on diffusive processes. A limitation that occur for spherical space is that it has a finite surface, limiting the largest geodesic path. For the hyperbolic space, having large curvature, small R, causes large growth. This causes numerical instability for our simulation and fails our threshold requirement for the expansion, hence the time condition for small curvature.

2.3.4 Results: Lévy and Ballistic Case

In the case of Brownian motion, section §2.3.3, we showed the curvature effects for diffusive processes. Intrinsic properties start to affect random walks, in which our MSD expansion, equation (2.8), accounts for. As curvature decreases, our results start to resemble Euclidean behavior. Therefore, our simulation agrees with the theoretical assumption. This section shifts our focus to Lévy and ballistic motion instead of Brownian for spherical space. In the original derivation of our expansion, [41], they considered only Brownian motion. Therefore, we are not confident that the same coefficient relation with curvature will work for non-Brownian motion. The third order term had no significant contribution and is excluded for the remainder of our work. Our objective is to show that our expansion still works but the curvature correction terms have a dependence on the Lévy exponent μ and as we decrease curvature, we get Euclidean-like behaviors. We expect the anomalous exponent relation from equation (2.3) to remain valid.

The presented results are for simulations in spherical space with $R = 10, 100$.

Trajectories are produced with Lévy exponents $\mu = 1.6, 2.4$ for 100 samples, shown in Figure 2.5. Using equation (2.99) to compute the MSD, the results are plotted with red-dashed line, with its 95% confidence interval as the gray shaded regions. Euclidean results are shown with black-dashed line with the yellow region as its 95% confidence interval. In this case, the spherical space MSD is fitted to equation (2.9d) shown in magenta, with the additional parameter, $\widetilde{\beta}_1$, to the second term accounting for curvature correction. In the case of Brownian motion, the coefficient equates to $\widetilde{\beta}_1 = \frac{4R^2}{3}$, i.e., with $R = 10, 100$, we have $\widetilde{\beta}_1 = 133, 13333$, respectively. We use this to gauge the impact of curvature for non-Brownian motion by computing the percentage difference from the fit.

Our simulation for non-Brownian motion in spherical space with radius $R = 10, 100$ for 100 samples, is shown in Figure 2.5. For Lévy motion, the simulation has Lévy exponent $\mu = 2.4$, shown in Figure 2.5a,b. In the ballistic case, Lévy exponent $\mu = 1.6$, shown in Figure 2.5c,d.

Starting with Lévy motion, for $R = 10$, we predict up to 24 time-steps where 75% of the data is used for the fit, shown in Figure 2.5a. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 1.601 \pm 0.01$ with an error of 0.06% from theory. The corresponding diffusion constant is $\widetilde{D} = 0.17 \pm 0.004$ with curvature coefficient of $\widetilde{\beta}_1 = 372 \pm 16.3$. Compared to Brownian motion, the fitted coefficient has a percentage difference of 180%. The Euclidean data remains above spherical and the deviation increases with time. To better understand the influence of curvature, we look at the percentage difference of the second order term to the first order term of equation (2.9d). The second order term has a percentage difference of $6.3\%, 25.6\%$ for time-steps $t = 10, 24$, respectively. As we increase the time, the curvature contribution increases significantly.

In the case of $R = 100$, Figure 2.5b, the prediction is for 100 time-steps in which 70% of the data is used for the fit with anomalous exponent of $\widetilde{\alpha} = 1.561 \pm 0.005$, which is within 2.4% from theory. The diffusion constant is $\widetilde{D} = 0.226 \pm 0.003$. The fitted coefficient is $\widetilde{\beta}_1 = 183000 \pm 10200$ with percent difference of 1273% from Brownian. Visually, Euclidean data still remains above spherical and the spread increases with time, but not as significant compared to $R = 10$. The second order term has a percentage difference from the first order of $4.6\%, 13.7\%$ for time-steps $t = 50, 100$, respectively. Curvature contribution increases significantly with time. The simulation of Lévy motion with our methodology reproduces the anomalous exponent relation

of $\alpha = 4 - \mu$. Compared to Brownian motion, Lévy increases the curvatures influence on random walks and the fitted curvature coefficients are no longer the same. In addition, as we decrease the curvature, the deviation from Euclidean also decreases.

Moving over to ballistic motion on a sphere. For $R = 10$, Figure 2.5c, the prediction is for 10 time-steps in which 70% of the data is used for the fit. The fitted MSD has an anomalous exponent of $\widetilde{\alpha} = 1.873 \pm 0.011$ with 6.4% error from theory. The diffusion constant is $\widetilde{D} = 0.156 \pm 0.002$ and coefficient $\widetilde{\beta}_1 = 420 \pm 55.4$ with percent difference of 216% from Brownian. The Euclidean data remains above spherical and the spread increases with time. Similar to the Lévy case, for time-steps $t = 5, 10$, the second order terms percentage difference form first order is $3.2\%, 12.1\%$, respectively. Thus, increasing time, increases curvature contribution.

For $R = 100$, we predict up to 50 time-steps where 70% of the data is used for the fit, shown in Figure 2.5d. The fit has anomalous exponent of $\widetilde{\alpha} = 1.901 \pm 0.003$, which is within 4.9% from theory and a diffusion constant of $\widetilde{D} = 0.171 \pm 0.002$. The fitted coefficient is $\widetilde{\beta}_1 = 107000 \pm 7910$ with percentage difference of 703% from Brownian. Visually, Euclidean data still remains above the spherical. Similarly to $R = 10$, curvature contribution increases with time, i.e., for time-steps $t = 25, 50$, the percentage difference from the second order to first order term is $2.1\%, 7.8\%$, respectively. Our simulation reproduces the anomalous exponent relation of $\alpha = 2$ for ballistic motion with increasing curvature influence on random walks. The fitted curvature coefficients are no longer the same as Brownian. Deviation from Euclidean also decreases as we decrease curvature.

Overall, compared to Brownian, we can still reproduce the results from equation (2.3) but the coefficients of our expansion term do not follow the scaled relation. As curvature decreases, the second order correction term contributes less, also observed in section §2.3.3.

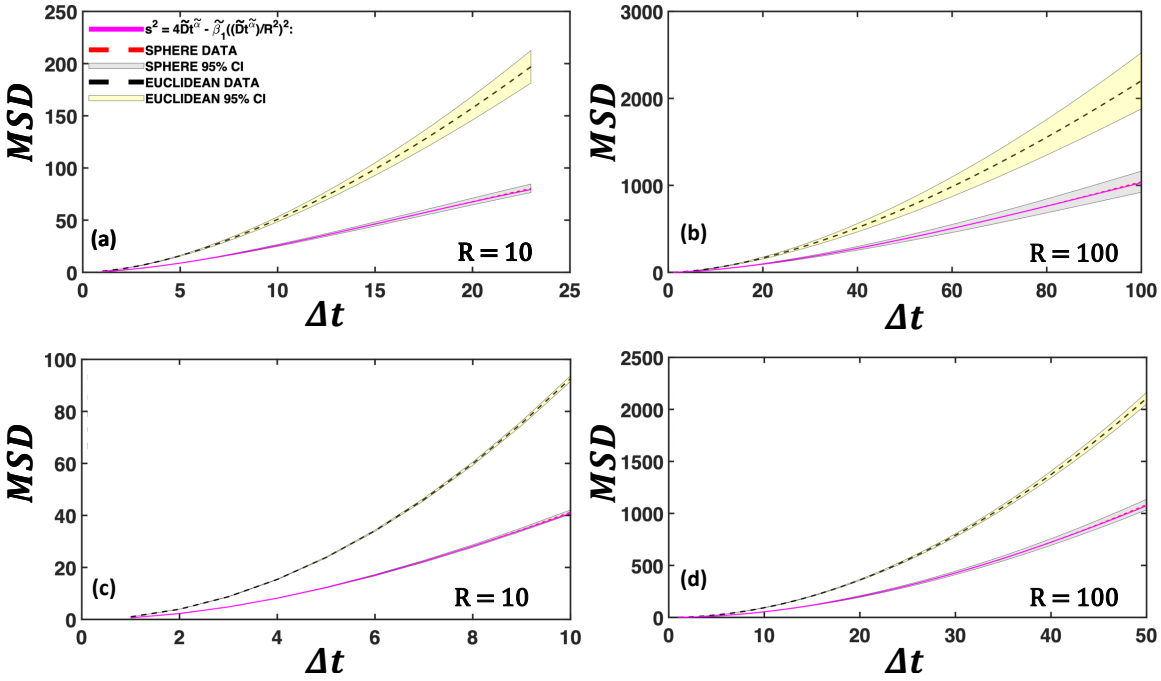

Figure 2.5: Shown are the MSD results for non-Brownian motion in spherical space for $R = 10$ and 100. Geodesic step lengths are drawn from equation (2.1). The simulation consists of 100 samples and the computed MSD are the red-dashed line and gray regions as the 95% confidence interval. A fit is performed on the simulation MSD, (red-dashes), by equation (2.9d), shown by the magenta lines. Euclidean results are the black-dashed line with the yellow region as its 95% confidence interval. (a,b) show Lévy motion with $\mu = 2.4$ and (c,d) for ballistic, with $\mu = 1.6$. (a) $R = 10$. We predict up to 24 time-steps in which 38% of the data is used for the fit. The fit has anomalous exponent of $\widetilde{\alpha} = 1.601 \pm 0.01$, with 0.06% error, diffusion constant of $\widetilde{D} = 0.17 \pm 0.004$ and a curvature coefficient of $\widetilde{\beta}_1 = 372 \pm 16.3$. (b) $R = 100$. We predict up to 100 time-steps with 70% of the data used for the fit. The fitted has anomalous exponent of $\widetilde{\alpha} = 1.561 \pm 0.005$, with 2.4% error, diffusion constant of $\widetilde{D} = 0.226 \pm 0.003$ and a coefficient of $\widetilde{\beta}_1 = 183000 \pm 10200$. Our simulation reproduces the expected anomalous exponent, $\alpha = 1.6$, for Lévy motion but the curvature contribution from the second term increases significantly with time, as explained in the main text. (c) $R = 10$. We predict up to 10 time-steps with 70% of the data used for the fit. The fitted MSD has anomalous exponent of $\widetilde{\alpha} = 1.873 \pm 0.011$, with 6.4% error, diffusion constant of $\widetilde{D} = 0.156 \pm 0.002$ and a coefficient of $\widetilde{\beta}_1 = 420 \pm 55.4$. (d) $R = 100$. Predicted for 50 time-steps in which 70% is used for the fit. The fit has anomalous exponent of $\widetilde{\alpha} = 1.901 \pm 0.003$, which is within 4.9% error, diffusion constant of $\widetilde{D} = 0.171 \pm 0.002$ and a coefficient of $\widetilde{\beta}_1 = 107000 \pm 7910$. In the case of ballistic motion, the simulation remains around the expected anomalous exponent, $\alpha = 2$. Overall, the deviation from Euclidean decreases as curvature decreases. Our methodology reproduces the results from equation (2.3) for non-Brownian motion.

2.4 Discussion

In this chapter, we have introduced a method of transportation on manifolds to study the effects of curvature for anomalous diffusion. We have developed a computational model to simulate random walks in spherical and hyperbolic spaces where objects are transported through geodesics. Furthermore, we have developed an analytical method that shows the dependence on curvature in MSD. The methodology was adapted to random walks in Euclidean, section §2.3.2, where our results remain within 5% error as theorized in equation (2.3).

We have shown that for small perturbations of geodesics, intrinsic effects occur for random walks in different curved spaces. We use Lévy distributions to produce our geodesic lengths which allow us to explore the full range of anomalous diffusion. For Brownian motion, $\mu = 3.4$, spherical and hyperbolic space does indeed behave as theorized from equation (2.3). However, additional terms are needed to account for curvature corrections. Therefore, based on the curvature of the space, transportation properties will not change given the accountability of small perturbations and curvature corrections. In the case of non-Brownian motion, section §2.3.4, Lévy and ballistic behaviors were also reproduced with similar results. These observations shed light on the inherent characteristics induced by various manifolds and quantify the curvature effects in different ways. For example, on a manifold with positive curvature, transportation is negatively impacted compared to a zero curvature manifold, such as a box. Shown in section §2.3.3, for Brownian motion, random walks performed on the sphere produce MSDs that deviate less when compared to the Euclidean space. Additionally, as we increase the curvature of the manifold, the curvature correction terms start to have stronger influence on MSD; whereas when we decrease curvature, section §2.3.3, they have less of an influence on MSD and start to converge to random walks in the Euclidean space. Our results are consistent with the work from reference [41], in the Brownian case.

More work is still needed to understand the effects of curvature. Our MSD expansion for anomalous diffusion have analytical coefficient terms that hold for Brownian motion but not for Lévy or ballistic motion. It would be interesting to restrict the hyperbolic spaces maximum geodesic length. This can be done by working with the Poincaré Disk model, and fixing a maximum distance from the center of it, which

would be the largest allowed geodesic length. This will allow us to quantify the maximum time threshold for the hyperbolic space. It will also be a more robust method to understand the relation of the expansion coefficients for non-Brownian motion on the sphere and hyperboloid. In the case of long time curvature effects on the system, random walks with different step lengths and time relationships could produce compelling phenomena. In our work, we used a ballistic relationship between path distance and time, equations (2.89, 2.92). Other relationship conditions could be explored that may shed light on properties not observed in our work.

Since we have a robust working method for Brownian motion, section §2.3.3, further work can include search processes on manifolds. For example, in the hyperbolic space, targets or resources could be spread out, uniformly, by performing a random walk using the Poincaré disk model then transforming those point to the half-plane. Uniform radial distribution can be defined as the circumference of a disk divided by the total disk area, derived in reference [32]. This will further allow searches to be performed on tree-like systems, such as b-ary trees, where b is the trees branching factor. Moreover, for the spherical space, an interesting problem would be having searchers to have some drag effect related to the mechanical properties of the medium in which they are moving in. This can bring a new relationship between path distance and time. In both spaces, this allows the study of search efficiency, possible network tuning properties for optimization purposes, and the relationship between the diffusion constant and target density and curvature.

In section §2.3, we pointed out as curvature decreases, the second order correction term contribution decreases. The same results were observed in section §2.3.3, implying that the curvature corrections may not have a dependence on the Lévy exponent μ and could be an artifact of something else. To better understand this, shown in Figure 2.6, are plots of $\frac{\widetilde{\beta_1}}{4R^2}$ versus μ, where $\widetilde{\beta}_1$ is from the fit equations (2.9d), and the actual $\widetilde{\alpha}$ versus μ with $\widetilde{\alpha}$ reported in section §2.3. Since we only have hyperbolic results for Lévy exponent $\mu = 3.4$, we plot these results against R as an inset. Shown in Figure 2.6a are plots of $\frac{\widetilde{\beta_1}}{4R^2}$ versus μ for the sphere with $R = 10, 100$, blue and orange. The inset are hyperboloid results with $\frac{\widetilde{\beta_1}}{4R^2}$ versus R, green. We see for $R = 100$, orange, within Lévy motion range ($2 < \mu < 3$), fluctuations do occur; whereas for larger curvature with $R = 10$, blue, results for $\mu < 3$ do not fluctuate much but are still larger than $\mu > 3$. For the hyperboloid, results remain around

the theoretical ($\frac{4}{3R^2}$) value. In the case of $R = 1000$, the large error occurs due to being close to Euclidean like behavior since curvature is small. In addition, shown in Figure 2.6b are plots of the actual $\widetilde{\alpha}$ versus μ for the sphere with $R = 10$, (blue), and 100, (orange), with the red dashes representing the theoretical anomalous exponent. The inset is the hyperboloid results with $\widetilde{\alpha}$ versus R in green. The spherical space fit for the anomalous exponent aligns with the theoretical results. For hyperbolic, the theoretical value is $\alpha = 1$, which the fits remain within 5% error. In Euclidean space, searches that are optimized for μ indicate that we should do an even more diffusive (super-diffusive or smaller μ) in spherical space and the opposite for hyperbolic.

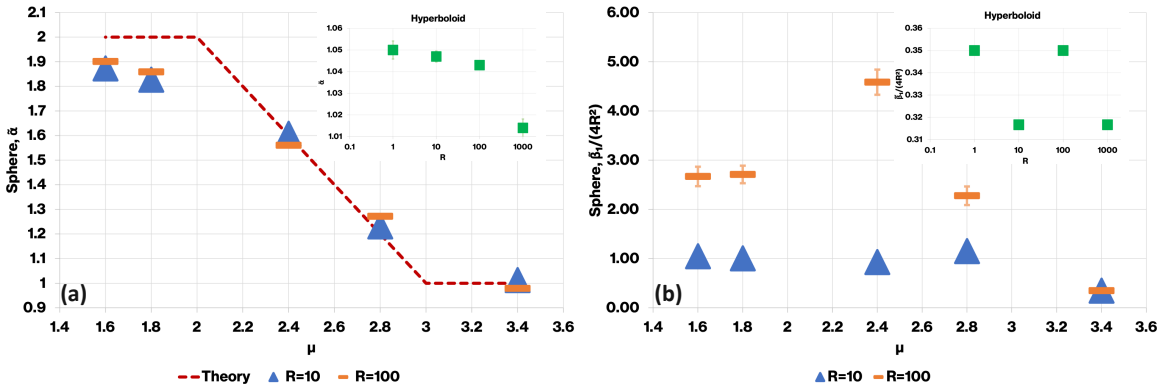

Figure 2.6: (a) Plots of the actual $\widetilde{\alpha}$ vs μ for the sphere with $R = 10$, (blue), and 100, (orange), where the red dashes represent the theoretical anomalous exponent. Inset: Hyperboloid results with $\widetilde{\alpha}$ vs R, green. (b) Plots of $\frac{\widetilde{\beta_1}}{4R^2}$ versus μ for the sphere with $R = 10$, (blue), and 100, (orange). The inset includes the corresponding hyperboloid results with $\frac{\widetilde{\beta_1}}{4R^2}$ vs R in green.

2.5 Appendix

Here, we start by going over the mathematics used for hyperbolic space. We parameterize the two-sheeted hyperboloid. Then apply stereographic projection of the hyperboloid to the Poincaré disk of radius R. Next, apply a Mőbius transformation of our disk to the half-plane. Then, we move to the derivation of hyperbolic geodesic equations, specifically for the half-plane model. Formulate the semi-circle relation for the half-plane model, used in our simulation. We then move to spherical space where we show the position, geodesic and central angle equations. Then introduce the method of quaternions, which is used to update positions on a sphere without

losing degrees of freedom. The final section goes over our simulation. We derive the appropriate geodesic Lévy relations for the given spaces, used in our random walk. Outline the simulation algorithm for each space and state the distance equation used to calculate the MSD for each space.

2.5.1 Mathematics

For a consistent flow in the derivation, we start by going over the various mathematics we use in the process of deriving the geodesics for hyperbolic space.

Stereographic Projection

Stereographic projection is a particular mapping (function) that projects a manifold onto a plane. It is conformal, meaning it preserves angles where the curves meet. It is neither isometric nor area-preserving: it preserves neither distances nor the areas of figures.

The general sketch of the stereographic projection is given in the form of a sphere. Consider a sphere in $\mathbb{R}^3$ of radius R. Let $N = (0, 0, R)$, the north pole of the sphere and assume M to be the set of all points on the sphere excluding N. We care for the intersection of the sphere with the plane at $z = 0$. For any point $P \in M$, there exist a unique line through N and P such that this line intersects the plane, $z = 0$, at one point P'. Let $y \in N$, the stereographic projection $\pi(x)$ of points $x \in M$ onto the plane at $z = 0$ has the following parametric form [42, 43].

$$\pi(x) = x + s(y - x) \tag{2.10}$$

The projection will be used to map points from the two-sheeted hyperboloid to the Poincaré disk.

Mőbius Transformation

Mőbius transformations is an isometric (distance preserving), one-to-one mapping. The transformation is a rational function of the following form,

$$f(z) = \frac{az + b}{cz + d} \tag{2.11}$$

where $z \in \mathbb{C}$ and with complex coefficients satisfying $ad - bc \neq 0$ [42]. For our purpose, this will preserve distance between the Poincaré disk and half-plane model.

First Fundamental Form

We start by stating the **First Fundamental Form** equation from differential geometry [42, 43].

$$ds^2 = Edu^2 + 2Fdudv + Gdv^2 \tag{2.12}$$

where $E(u,v) = \vec{x_u} \cdot \vec{x_u}$, $F(u,v) = \vec{x_u} \cdot \vec{x_v}$, and $G(u,v) = \vec{x_v} \cdot \vec{x_v}$ and $EG - F^2 \neq 0$. We can also modify the first fundamental form, equation (2.12), in a way such that the parameterization variables (u,v) has explicit dependence on length (s). We get the **Modified First Fundamental Form** as [42],

$$1 = E\left(\frac{du}{ds}\right)^2 + 2F\left(\frac{du}{ds}\right)\left(\frac{dv}{ds}\right) + G\left(\frac{dv}{ds}\right)^2 \tag{2.13}$$

A curve $\gamma(s) = \vec{x}(u(s), v(s))$ on a surface, parameterized by length (s), is a **geodesic** if and only if,

$$\frac{d}{ds}(E\acute{u} + F\acute{v}) = \frac{1}{2}(E_u\acute{u}^2 + 2F_u\acute{u}\acute{v} + G_u\acute{v}^2) \tag{2.14a}$$

$$\frac{d}{ds}(F\acute{u} + G\acute{v}) = \frac{1}{2}(E_v\acute{u}^2 + 2F_v\acute{u}\acute{v} + G_v\acute{v}^2) \tag{2.14b}$$

where $\acute{u} = \frac{du}{ds}$ and $\acute{v} = \frac{dv}{ds}$ [44]. The geodesic equations (2.14) will be used to find the relations for x, y and s that will give us the minimal path.

Euler-Lagrange Equations

To make sure we have the minimal path, we can also apply the Euler-Lagrange equation from calculus of variations.

Given a functional of the form,

$$J(\vec{q}) = \int \mathcal{L}(t; \vec{q}, \dot{\vec{q}})dt$$

where $\dot{\vec{q}} = \frac{d\vec{q}}{dt}$. The functional $J(\vec{q})$ is referred to as the action function. Applying the

local minima conditions [45], we get the **Euler-Lagrange** equations as,

$$\frac{\partial \mathcal{L}}{\partial \vec{q}} - \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{\vec{q}}}\right) = 0 \tag{2.15}$$

Note that if $\frac{\partial \mathcal{L}}{\partial t} = 0$, then $\mathcal{L}(t; \vec{q}, \dot{\vec{q}}) = \mathcal{L}(\vec{q}, \dot{\vec{q}})$, and we get the **Modified Euler-Lagrange Equations** as,

$$\mathcal{L}(\vec{q}, \dot{\vec{q}}) - \dot{\vec{q}}\frac{\partial \mathcal{L}(\vec{q}, \dot{\vec{q}})}{\partial \dot{\vec{q}}} = C \tag{2.16}$$

where C is an arbitrary constant.

2.5.2 Hyperbolic Geometry

This section derives the required equations for hyperbolic geodesic, specifically for the half-plane model and formulates the semi-circle relation for the half-plane, used in our simulation. We start by parameterizing the two-sheeted hyperboloid, then mapping it to the Poincaré disk along with the inverse map. The same will be done for the hyperbolic half-plane model for completeness. In the process, certain geometric objects (metric, distance, area) will be shown, specific for that model, which will be used in later derivations.

Two-Sheeted Hyperboloid

Here, we define the two-sheeted hyperboloid, prove its parameterization and give the metric and distance equation. The **two-sheeted hyperboloid** is defined as the following,

$$\mathbb{H}^2_R \in \left\{(x, y, z) \in \mathbb{R} \mid x^2 + y^2 - z^2 = -R^2\right\} \tag{2.17}$$

We start by parameterizing the **two-sheeted hyperboloid** where we only consider the upper sheet, $z(u, v) > 0$.

$$x(u, v) = R\sinh(u)\cos(v) \tag{2.18a}$$

$$y(u, v) = R\sinh(u)\sin(v) \tag{2.18b}$$

$$z(u, v) = R\cosh(u) \tag{2.18c}$$

where $u \in [0, \infty)$ and $v \in [0, 2\pi]$.

Proof. **Two-Sheeted Hyperboloid Parameterization**

Consider the following parameterization:

$$\begin{aligned} x(r, v) &= r\cos(v) \\ y(r, v) &= r\sin(v) \\ z(r, v) &= z \end{aligned}$$

where (r, v) are parameters with $r^2 = x^2 + y^2$ and $v \in [0, 2\pi]$. Plugging this into equation (2.17) yields the surface equation of our hyperboloid:

$$x^2 + y^2 - z^2 = r^2 - z^2 = -R^2$$

From here, apply the hyperbolic trigonometry relation $\Big(\cosh^2(u) - \sinh^2(u) = 1\Big)$, with u being a new parametrization variable. Multiplying the relation by R^2, to give $R^2\cosh^2(u) - R^2\sinh^2(u) = R^2$. Our new relation is the following:

$$\begin{aligned} R^2 &= z^2 - r^2 \\ R^2 &= R^2\cosh(u)^2 - R^2\sinh(u)^2 \end{aligned}$$

By comparison and with $u \in [0, \infty)$:

$$\begin{aligned} z(u) &= \pm R\cosh(u) \\ r(u) &= \pm R\sinh(u) \end{aligned}$$

By this convention and choosing the upper sheet, $z \in [R, \infty)$ and $r \in [0, \infty)$, we have our two-sheeted hyperboloid parameterization with $u \in [0, \infty)$ and $v \in [0, 2\pi]$.

$$\begin{aligned} x(u, v) &= R\sinh(u)\cos(v) \\ y(u, v) &= R\sinh(u)\sin(v) \\ z(u, v) &= R\cosh(u) \end{aligned}$$

□

In addition to the two-sheeted hyperboloids parameterized equation (2.18), the metric for our hyperboloid is defined as,

$$ds^2_{\mathbb{H}^2_R}(x,y,z) = dx^2 + dy^2 - dz^2 \tag{2.19a}$$

$$ds^2_{\mathbb{H}^2_R}(u,v) = R^2 du^2 + R^2 \sinh(u)^2 dv^2 \tag{2.19b}$$

The hyperbolic distance between two points on the hyperboloid is also stated.

$$d_{\mathbb{H}^2_R} = R \operatorname{arcCosh}\left(\frac{z_1 z_2 - x_1 x_2 - y_1 y_2}{R^2}\right) \tag{2.20}$$

Poincaré Disk Model

Here, we define the Poincaré disk model, show its parameterization, give the metric, show distance equation along with the circumference and area. Then show the mapping from the hyperboloid to the disk and its corresponding inverse. The **Poincaré disk** model is defined as the following,

$$\mathbb{D}_R \in \left\{(x,y) \in \mathbb{R} \mid x^2 + y^2 < R^2\right\} \tag{2.21}$$

The disk can be represented by the given parameterization,

$$x(r_E, \theta) = r_E \cos(\theta) \tag{2.22a}$$

$$y(r_E, \theta) = r_E \sin(\theta) \tag{2.22b}$$

where $r_E \in [0, R)$ and $\theta \in [0, 2\pi]$. Note that r_E is the Euclidean radial distance from the center of the disk.

The metric for the Poincaré disk model is defined as:

$$ds^2_{\mathbb{D}_R}(x,y) = \frac{4R^4}{(R^2 - x^2 - y^2)^2}\left(dx^2 + dy^2\right) \tag{2.23a}$$

$$ds^2_{\mathbb{D}_R}(r_E,\theta) = \frac{4R^4}{(R^2 - r_E^2)^2}\left(dr_E^2 + r_E^2 d\theta^2\right) \tag{2.23b}$$

The hyperbolic distance for the Poincaré disk model can be stated in two ways.

First, the distance from the origin of this disk.

$$d_{\mathbb{D}_R} = r_{\mathbb{D}_R} = 2R\,\text{arcTanh}\left(\frac{r_E}{R}\right) = R\ln\left(\frac{R+r_E}{R-r_E}\right) \tag{2.24}$$

With $r_{\mathbb{D}_R} \in [0,\infty)$. This gives the relationship between the Poincaré disk model to the Euclidean disk of Radius R. Second, the distance between two points in the Poincaré disk.

$$d_{\mathbb{D}_R} = R\,\text{arcCosh}\left(1 + \frac{2R^2\big((x_2-x_1)^2+(y_2-y_1)^2\big)}{(R^2-x_1^2-y_1^2)(R^2-x_2^2-y_2^2)}\right) \tag{2.25}$$

We can also rewrite the Poincaré disk metric in polar form using $r_{\mathbb{D}_R}$ as the radius.

$$ds^2_{\mathbb{D}_R}(r_{\mathbb{D}_R},\theta) = \left(dr^2_{\mathbb{D}_R} + R^2\left(\sinh\left(\frac{r_{\mathbb{D}_R}}{R}\right)\right)^2 d\theta^2\right) \tag{2.26}$$

For completeness, we also cover the derivation of the circumference and area of the Poincaré disk. By using its metric, equation (2.23), we get the following equations.

$$C_{\mathbb{D}_R}(r_{\mathbb{D}_R}) = \int_0^{2\pi} \frac{2R^2 r_E}{R^2-r_E^2}\,d\theta = \frac{4\pi R^2 r_E}{R^2-r_E^2} = 2\pi R\sinh\left(\frac{r_{\mathbb{D}_R}}{R}\right) \tag{2.27}$$

$$A_{\mathbb{D}_R}(r_{\mathbb{D}_R}) = \int_0^{r(r_{\mathbb{D}_R})}\int_0^{2\pi} \frac{4R^4}{(R^2-r_E^2)^2} r_E\,dr_E\,d\theta = 2\pi R^2\left[\cosh\left(\frac{r_{\mathbb{D}_R}}{R}\right)-1\right] \tag{2.28}$$

where r_E is the Euclidean radius and can be represented as a function of the Poincaré disk distance by taking the inverse of equation (2.24) $\left(r_E = R\tanh\left(\frac{r_{\mathbb{D}_R}}{2R}\right)\right)$. Comparing the Poincaré disk equations (2.27) and (2.28) to the Euclidean, $\Big(C(r_E) = 2\pi r_E\Big)$ and $\Big(A(r_E) = \pi r_E^2\Big)$, objects in Euclidean grow as polynomials; whereas in hyperbolic, they grow as exponentials.

We now apply steroegraphic projection to get the relation between points on the hyperboloid to points in the Poincaré disk model, $\mathbb{H}^2_R \to \mathbb{D}_R$. Let $\Big(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}\Big) \in \mathbb{D}_R$ and $\Big(x_{\mathbb{H}^2_R}, y_{\mathbb{H}^2_R}, z_{\mathbb{H}^2_R}\Big) \in \mathbb{H}^2_R$. We want to project the two-sheeted hyperbolid $(x^2_{\mathbb{H}^2_R} + y^2_{\mathbb{H}^2_R} - z^2_{\mathbb{H}^2_R} = -R^2)$ onto a disk $(x^2_{\mathbb{D}_R} + y^2_{\mathbb{D}_R} < R)$, such that the one-to-one mapping is conformal. Since the projection is of a $3-$dimensional object into

a 2−dimensional disk, we let $z_{\mathbb{D}_R} = 0$. The projection is from the point $(0, 0, -R)$. Following the method in section §2.5.1, we get the following,

$$(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}, z_{\mathbb{D}_R}) = (t \cdot x_{\mathbb{H}^2_R}, t \cdot y_{\mathbb{H}^2_R}, -R + t \cdot (z_{\mathbb{H}^2_R} + R))$$

where t is a scalar. Applying $z_{\mathbb{D}_R} = 0$ and solving for t, we get the scalar $t = \frac{R}{z_{\mathbb{H}^2_R}+R}$. The final form of the relation is shown.

$$x_{\mathbb{D}_R} = \frac{R}{z_{\mathbb{H}^2_R} + R} \cdot x_{\mathbb{H}^2_R} \tag{2.29a}$$

$$y_{\mathbb{D}_R} = \frac{R}{z_{\mathbb{H}^2_R} + R} \cdot y_{\mathbb{H}^2_R} \tag{2.29b}$$

$$z_{\mathbb{D}_R} = 0 \tag{2.29c}$$

We can also compute the inverse mapping, $\mathbb{D}_R \to \mathbb{H}^2_R$. Let $\left(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}\right) \in \mathbb{D}_R$ and $\left(x_{\mathbb{H}^2_R}, y_{\mathbb{H}^2_R}, z_{\mathbb{H}^2_R}\right) \in \mathbb{H}^2_R$. In this case, we project points on the disk onto our two-sheeted hyperboloid. Applying the same method in section §2.5.1, we have the following,

$$(x_{\mathbb{H}^2_R}, y_{\mathbb{H}^2_R}, z_{\mathbb{H}^2_R}) = (s \cdot x_{\mathbb{D}_R}, s \cdot y_{\mathbb{D}_R}, -R + s \cdot (z_{\mathbb{D}_R} + R))$$

where s is a scalar and $z_{\mathbb{D}_R} = 0$ still applies here. The projection is mapped to a manifold of higher dimension such that we have to use the relation, $\left(x^2_{\mathbb{H}^2_R} + y^2_{\mathbb{H}^2_R} - z^2_{\mathbb{H}^2_R} = -R^2\right)$, to solve for the scalar s. Plugging the relations in and applying the quadratic root equation, $s = \frac{2R^2}{R^2 - x^2_{\mathbb{D}_R} - y^2_{\mathbb{D}_R}}$, we get the final relation as follows,

$$x_{\mathbb{H}^2_R} = \frac{2R^2}{R^2 - x^2_{\mathbb{D}_R} - y^2_{\mathbb{D}_R}} \cdot x_{\mathbb{D}_R} \tag{2.30a}$$

$$y_{\mathbb{H}^2_R} = \frac{2R^2}{R^2 - x^2_{\mathbb{D}_R} - y^2_{\mathbb{D}_R}} \cdot y_{\mathbb{D}_R} \tag{2.30b}$$

$$z_{\mathbb{H}^2_R} = R \cdot \frac{R^2 + x^2_{\mathbb{D}_R} + y^2_{\mathbb{D}_R}}{R^2 - x^2_{\mathbb{D}_R} - y^2_{\mathbb{D}_R}} \tag{2.30c}$$

A key component to point out here, from our sterographic mapping, is the relation between the Poincaré disk radial distance $r_{\mathbb{D}_R} \in [0, \infty)$, equation (2.24), to the

parameter $u \in [0, \infty)$ in equation (2.18), shown in equation (2.31). The parameter u can be thought of as a dimensionless representation of the Poincaré disks radial distance. Moreover, it shows that the stereographic mapping is not isometric, thus distance is not preserved but the angles are.

$$u\left(r_{\mathbb{D}_R}\right) = \frac{r_{\mathbb{D}_R}}{R} \tag{2.31}$$

Hyperbolic Half-Plane Model and Geodesic Equation

Here, we define the hyperbolic half-plane model, give the metric and distance equation. Then show the mapping from and to the half-plane and the Poincaré disk and then for the hyperboloid.

The hyperbolic **half-plane** model is defined such that we consider only the upper part of a plane, $y > 0$.

$$\mathbb{H}_R \in \left\{(x, y) \in \mathbb{R} \mid y > 0\right\} \tag{2.32}$$

The corresponding metric for this model is defined as,

$$ds^2_{\mathbb{H}_R}(x, y) = \frac{R^2}{y^2}\left(dx^2 + dy^2\right) \tag{2.33}$$

The distance between two points using the half-plane model has the following form,

$$d_{\mathbb{H}_R} = R\,\text{arcCosh}\left(\frac{(x_2 - x_1)^2 + (y_2^2 + y_1^2)}{2y_1y_2}\right) \tag{2.34}$$

To better understand the half-plane, we next move to mapping it to the other models. We apply a Mőbius transformation to get the relation between points on the half-plane to points on the Poincaré disk, $\mathbb{H}_R \to \mathbb{D}_R$. Let $\left(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}\right) \in \mathbb{D}_R$ and $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right) \in \mathbb{H}_R$. Additionally, let $\left(w, z\right) \in \mathbb{C}$ such that $w = x_{\mathbb{H}_R} + iy_{\mathbb{H}_R}$ represents points in the half-plane and $z = x_{\mathbb{D}_R} + iy_{\mathbb{D}_R}$, representing points in the Poincaré disk. Working with the Mőbius transformation, presented in section §2.5.1, we get the following map between z and w,

$$z = R \cdot \frac{-w + iR}{w + iR} \tag{2.35}$$

By applying the complex conjugate to the right side of equation (2.35) and plugging the component terms in we get the following relation,

$$z = \frac{R}{x_{\mathbb{H}_R}^2 + (R + y_{\mathbb{H}_R})^2}\left((R^2 - x_{\mathbb{H}_R}^2 - y_{\mathbb{H}_R}^2) + i(2R \cdot x_{\mathbb{H}_R})\right)$$

In component form, we get our final mapping from the half-plane to the Poincaré disk.

$$x_{\mathbb{D}_R} = \Re(z) = \frac{R}{x_{\mathbb{H}_R}^2 + (R + y_{\mathbb{H}_R})^2}\left(R^2 - x_{\mathbb{H}_R}^2 - y_{\mathbb{H}_R}^2\right) \tag{2.36a}$$

$$y_{\mathbb{D}_R} = \Im(z) = \frac{R}{x_{\mathbb{H}_R}^2 + (R + y_{\mathbb{H}_R})^2}\left(2R \cdot x_{\mathbb{H}_R}\right) \tag{2.36b}$$

The inverse is also shown, $\mathbb{D}_R \to \mathbb{H}_R$. Let $\left(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}\right) \in \mathbb{D}_R$ and $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right) \in \mathbb{H}_R$. Let $z = x_{\mathbb{D}_R} + iy_{\mathbb{D}_R}$ and $w = x_{\mathbb{H}_R} + iy_{\mathbb{H}_R}$ as before. Applying the inverse of equation (2.35), we have the following relation,

$$w = R \cdot \frac{i(R - z)}{R + z} \tag{2.37}$$

Again, by applying the complex conjugate to the right side of equation (2.37) and plugging the component terms in we get the following relation,

$$w = \frac{R}{y_{\mathbb{D}_R}^2 + (R + x_{\mathbb{D}_R})^2}\left((2R \cdot y_{\mathbb{D}_R}) + i(R^2 - x_{\mathbb{D}_R}^2 - y_{\mathbb{D}_R}^2)\right)$$

In component form, we have the final relation from points in the Poincaré disk to the half-plane.

$$x_{\mathbb{H}_R} = \Re(w) = \frac{R}{y_{\mathbb{D}_R}^2 + (R + x_{\mathbb{D}_R})^2}\left(2R \cdot y_{\mathbb{D}_R}\right) \tag{2.38a}$$

$$y_{\mathbb{H}_R} = \Im(w) = \frac{R}{y_{\mathbb{D}_R}^2 + (R + x_{\mathbb{D}_R})^2}\left(R^2 - x_{\mathbb{D}_R}^2 - y_{\mathbb{D}_R}^2\right) \tag{2.38b}$$

Note, the half-plane model and Poincaré disk are not conformal, but isometric.

We move over to the half-plane and the hyperboloid and show its mapping. Let $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right) \in \mathbb{H}_R$ and $\left(x_{\mathbb{H}_R^2}, y_{\mathbb{H}_R^2}, z_{\mathbb{H}_R^2}\right) \in \mathbb{H}_R^2$. We want to project the two-sheeted hyperbolid ($x_{\mathbb{H}_R^2}^2 + y_{\mathbb{H}_R^2}^2 - z_{\mathbb{H}_R^2}^2 = -R^2$) into the half-plane, ($y_{\mathbb{H}_R} > 0$). We utilize the

hyperboloid relation to the Poincaré disk, equations (2.29), and apply it to the Half-Plane equations (2.38). Before doing so, let $\left(x_{\mathbb{D}_R}, y_{\mathbb{D}_R}\right) \in \mathbb{D}_R$ and combine terms together in equation (2.29). Working through the algebra, we have the following terms,

$$\begin{aligned} x_{\mathbb{D}_R}^2 + y_{\mathbb{D}_R}^2 &= \frac{R}{\left(R + z_{\mathbb{H}_R^2}\right)^2}\left(x_{\mathbb{H}_R^2}^2 + y_{\mathbb{H}_R^2}^2\right) \\ y_{\mathbb{D}_R}^2 + \left(x_{\mathbb{D}_R} + R\right)^2 &= \frac{2R^2}{\left(R + z_{\mathbb{H}_R^2}\right)^2}\left(z_{\mathbb{H}_R^2} + x_{\mathbb{H}_R^2}\right) \\ R^2 - x_{\mathbb{D}_R}^2 - y_{\mathbb{D}_R}^2 &= \frac{2R^3}{\left(R + z_{\mathbb{H}_R^2}\right)} \end{aligned}$$

By plugging the algebraic relations into equations (2.38), we get the mapping from the hyperboloid to the half-plane.

$$x_{\mathbb{H}_R} = \frac{R}{z_{\mathbb{H}_R^2} + x_{\mathbb{H}_R^2}} y_{\mathbb{H}_R^2} \tag{2.39a}$$

$$y_{\mathbb{H}_R} = \frac{R^2}{z_{\mathbb{H}_R^2} + x_{\mathbb{H}_R^2}} \tag{2.39b}$$

$$\tag{2.39c}$$

We can also go the other way around by computing the inverse, $\mathbb{H}_R \to \mathbb{H}_R^2$. Let $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right) \in \mathbb{H}_R$ and $\left(x_{\mathbb{H}_R^2}, y_{\mathbb{H}_R^2}, z_{\mathbb{H}_R^2}\right) \in \mathbb{H}_R^2$. In this case, we map the half-plane onto our hyperboloid. Working with the hyperbolic sheet equation (2.17) and applying it to equation (2.39) and taking the square, we get the following algebraic relations,

$$\begin{aligned} z_{\mathbb{H}_R^2} + x_{\mathbb{H}_R^2} = \frac{R}{x_{\mathbb{H}_R}} y_{\mathbb{H}_R^2} = \frac{R^2}{y_{\mathbb{H}_R}} &\rightarrowtail y_{\mathbb{H}_R^2} = R\frac{x_{\mathbb{H}_R}}{y_{\mathbb{H}_R}} \\ z_{\mathbb{H}_R^2} &= \frac{R^2}{y_{\mathbb{H}_R}} - x_{\mathbb{H}_R^2} \\ x_{\mathbb{H}_R^2}^2 - z_{\mathbb{H}_R^2}^2 &= -R^2 - y_{\mathbb{H}_R^2}^2 \\ z_{\mathbb{H}_R^2}^2 &= \frac{R^4}{y_{\mathbb{H}_R}^2} + x_{\mathbb{H}_R^2}^2 - 2x_{\mathbb{H}_R^2}\frac{R^2}{y_{\mathbb{H}_R}} \end{aligned}$$

We have an expression for $y_{\mathbb{H}^2_R}$, which allows us to solve for $x_{\mathbb{H}^2_R}$ and $z_{\mathbb{H}^2_R}$. After substitution and some algebra, our final mapping from the half-plane to the hyperboloid is the following,

$$x_{\mathbb{H}^2_R} = \frac{1}{2y_{\mathbb{H}_R}}\left(R^2 - x^2_{\mathbb{H}_R} - y^2_{\mathbb{H}_R}\right) \tag{2.40a}$$

$$y_{\mathbb{H}^2_R} = R\frac{x_{\mathbb{H}_R}}{y_{\mathbb{H}_R}} \tag{2.40b}$$

$$z_{\mathbb{H}^2_R} = \frac{1}{2y_{\mathbb{H}_R}}\left(R^2 + x^2_{\mathbb{H}_R} + y^2_{\mathbb{H}_R}\right) \tag{2.40c}$$

When working with the different model for hyperbolic geometry, we use these derived equations as our mapping from one model to the other.

Next, we derive the geodesic equations for the half-plane by working with its metric equation (2.33). We form the Lagrange equation by taking the square root of equation (2.33) to properly form our action function, with $y' = \frac{dy}{dx}$.

$$\mathcal{L}(x; y, y') = \frac{R}{y}\sqrt{1 + y'^2} \tag{2.41}$$

Solving for the terms in the first fundamental form equation (2.12), we have the following relations,

$$E(x,y) = \frac{R^2}{y^2} \qquad F(x,y) = 0 \qquad G(x,y) = \frac{R^2}{y^2}$$

$$E_x(x,y) = 0 \qquad F_x(x,y) = 0 \qquad G_x(x,y) = 0$$

$$E_y(x,y) = -2\frac{R^2}{y^3} \qquad F_y(x,y) = 0 \qquad G_y(x,y) = -2\frac{R^2}{y^3}$$

Next, we plug our relation terms into the first geodesic condition equation (2.14a) to get the following differential equation,

$$\frac{d}{ds}\left(\frac{R^2}{y^2}\frac{dx}{ds}\right) = 0$$

Solving for $\frac{dx}{ds}$, we have the following half-plane **x-slope** relation,

$$\frac{dx}{ds} = \frac{C}{R^2}y^2 \tag{2.42}$$

From the **x-slope** relation, equation (2.42), we have two cases to consider for the constant C. First, when $C = 0$ and second for $C \neq 0$. Additionally, by applying our action function $\mathcal{L}(x; y, y')$, equation (2.41), to the Euler-Lagrange equation (2.16), since $\frac{\partial \mathcal{L}}{\partial x} = 0$, we have,

$$\frac{R}{y}\frac{1}{\sqrt{1 + (\frac{dy}{dx})^2}} = C$$

Solving for dx, we get our differential relation of $x(y)$ for the half-plane.

$$dx = \frac{y}{\sqrt{\frac{R^2}{C^2} - y^2}}dy \tag{2.43}$$

Plugging the **x-slope** relation, equation (2.42), into the modified first fundamental form, equation (2.13), we get the relation for the geodesic length s and spatial parameter y as follows,

$$1 = \frac{C^2}{R^2}y^2 + \frac{R^2}{y^2}\left(\frac{dy}{ds}\right)^2$$

Solving for ds, we get the differential relation for the geodesic $s(y)$ for the half-plane.

$$ds = \frac{R^2}{C}\frac{1}{y\sqrt{\frac{R^2}{C^2} - y^2}}dy \tag{2.44}$$

To solve these equations ((2.43), (2.44)), we have to consider the two conditions for the constant C. We approach the solution by keeping the differential forms positive and letting the integration bounds deal with the sign orientation, therefore $C \geq 0$.

• Case 1: $\boxed{C = 0}$

We start by working with the **x-slope** equation (2.42) and letting $C = 0$. Then $\frac{dx}{ds} = 0$. Plugging this into the modified first fundamental form equation (2.13), we

get the differential relation as follows,

$$ds = \frac{R}{y}dy$$

Solving the differential equation gives the geodesic $s(y)$ equations.

$$\Delta s(y) = R\ln(\frac{y}{y_0}) \tag{2.45}$$

Putting all the equations together and taking the inverse of the geodesic equation (2.45), we have the following geodesic equations for **Case** 1, which are **vertical lines in the half-plane**.

$$x_{2\mathbb{H}_R}(\Delta s) = x_{1\mathbb{H}_R}(\Delta s) \tag{2.46a}$$

$$y_{2\mathbb{H}_R}(\Delta s) = y_{1\mathbb{H}_R}(\Delta s)\, e^{\left(\frac{\Delta s}{R}\right)} \tag{2.46b}$$

$$\Delta s\left(y_{1\mathbb{H}_R}, y_{2\mathbb{H}_R}\right) = R\ln\left(\frac{y_{2\mathbb{H}_R}}{y_{1\mathbb{H}_R}}\right) \tag{2.46c}$$

In the vertical line case for the half-plane, we have the following useful relationship. If $y_{2\mathbb{H}_R} > y_{1\mathbb{H}_R}$, then $\Rightarrow \Delta s > 0$. If $y_{2\mathbb{H}_R} < y_{1\mathbb{H}_R}$, then $\Rightarrow \Delta s < 0$.

• Case 2: $\boxed{C \neq 0 \;\&\; C > 0}$

In this case, we start with the **x-slope** relation, equation (2.42), where $C \neq 0$. First we solve the $x(y)$ differential relation, equation (2.43), by elementary integration. We let $\Delta x = x - x_c$ to get the following $x(y)$ equation,

$$x(y) = x_c - \sqrt{\frac{R^2}{C^2} - y^2} \tag{2.47}$$

Now, solving to get $\frac{R}{C}$ by itself gives,

$$(\Delta x)^2 + y^2 = \frac{R^2}{C^2} \tag{2.48}$$

We see that when $C \neq 0$, we get our geodesic for the half-plane to be a semi-circle centered around x_c with radius $\frac{R}{C}$. Next, we solve the $s(y)$ differential relation by integrating equation (2.44). By letting $\Delta s = s_2 - s_1$, with $s_1 = 0$ corresponding to $y = \frac{R}{C}$ and choosing $\Delta s = s_2 = s(y)$ to be the total geodesic length. We then get the

following $s(y)$ equation.

$$s(y) = -R \ln \left(\frac{1}{y} \left(\frac{R}{C} + \sqrt{\frac{R^2}{C^2} - y^2} \right) \right) \tag{2.49}$$

Putting it all together, we take the inverse of equation (2.49) to get $y(s)$. Similarly, in equation (2.47), we replace the term $-\sqrt{\frac{R^2}{C^2} - y^2}$ by reorganization of equation (2.49), replacing y with the new form $y(s)$ and simplifying to get $x(s)$. We have the following geodesic equations for **Case** 2, which are **semi-circles centered about x_c in the half-plane**.

$$x(s) = x_c + \frac{R}{C} \tanh \left(\frac{s}{R} \right) \tag{2.50a}$$

$$y(s) = \frac{R}{C} \operatorname{sech} \left(\frac{s}{R} \right) \tag{2.50b}$$

$$s(y) = s(x, y) = -R \ln \left(\frac{1}{y} \left(\frac{R}{C} + \sqrt{\frac{R^2}{C^2} - y^2} \right) \right) \tag{2.50c}$$

Further, solving for C in equation (2.47) and assuming $C > 0$, we get the following form,

$$C(x, y) = \sqrt{\frac{R^2}{(x - x_c)^2 + y^2}} \tag{2.51}$$

We require $(x_1 - x_c)^2 + (y_1)^2 = \frac{R^2}{C^2}$ and $(x_2 - x_c)^2 + (y_2)^2 = \frac{R^2}{C^2}$. If (x_1, y_1) and (x_2, y_2) are on the same semi-circle, then they must have same radius $\frac{R}{C}$, allowing the following relation to hold,

$$(x_1 - x_c)^2 + (y_1)^2 = (x_2 - x_c)^2 + (y_2)^2$$

Therefore, solving for x_c, we get the **half-plane semi-circle center equation** as follows,

$$x_c = \frac{\left((x_2)^2 - (x_1)^2\right) + \left((y_2)^2 - (y_1)^2\right)}{2\,(x_2 - x_1)} \tag{2.52}$$

This implies both (x_1, y_1) and (x_2, y_2) have the same C. Allowing us to solve equation (2.51) for the value of C with any of the two points corresponding to the

same geodesic path. With the derived equations for both the vertical line and semi-circle cases, we can compute the geodesic of any path for the half-plane. The final forms of our geodesic equations for the half-plane model, with $r = \frac{R}{C}$, are,

$$x_c = \frac{\left((x_{2\mathbb{H}_R})^2 - (x_{1\mathbb{H}_R})^2\right) + \left((y_{2\mathbb{H}_R})^2 - (y_{1\mathbb{H}_R})^2\right)}{2\left(x_{2\mathbb{H}_R} - x_{1\mathbb{H}_R}\right)} \tag{2.53a}$$

$$r = \frac{R}{C} = \sqrt{\left(x_{1\mathbb{H}_R} - x_c\right)^2 + \left(y_{1\mathbb{H}_R}\right)^2} \tag{2.53b}$$

$$s_1\left(y_{1\mathbb{H}_R}\right) = -R\ln\left(\frac{1}{y_{1\mathbb{H}_R}}\left(r + \sqrt{(r)^2 - \left(y_{1\mathbb{H}_R}\right)^2}\right)\right) \tag{2.53c}$$

$$\Delta s\left(y_{1\mathbb{H}_R}, y_{2\mathbb{H}_R}\right) = s_2 - s_1 = -R\ln\left[\frac{y_{1\mathbb{H}_R}}{y_{2\mathbb{H}_R}}\left(\frac{r + \sqrt{(r)^2 - \left(y_{2\mathbb{H}_R}\right)^2}}{r + \sqrt{(r)^2 - \left(y_{1\mathbb{H}_R}\right)^2}}\right)\right] \tag{2.53d}$$

$$x_{2\mathbb{H}_R}(\Delta s) = x_c + r\cdot\tanh\left(\frac{s_1 + \Delta s}{R}\right) \tag{2.53e}$$

$$y_{2\mathbb{H}_R}(\Delta s) = r\cdot\operatorname{sech}\left(\frac{s_1 + \Delta s}{R}\right) \tag{2.53f}$$

Note, in equations (2.53), when applying equations (2.53e) and (2.53f), one needs to find x_c and Δs from other methods.

Half-Plane Locally Euclidean, Uniform Angle

Here, we show that the half-plane model is locally Euclidean, by applying a local approximation to points in the half-plane, allowing us to choose angles from a uniform distribution. The work here will be used to develop a relationship such that we can treat the semi-circle center x_c, from equation (2.52), as a random variable for our simulation. We begin by stating what the angular probability density function (PDF) in the half-plane is locally, as a function of angle θ.

$$\rho_{\mathbb{H}_R}(\theta) = \frac{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}}\right)\delta\cos(\theta) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right)\delta\sin(\theta)}{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)} \tag{2.54}$$

Proof. **Half-Plane Angular Probability Density Function**

To show $\rho(\theta)$ is the localized probability density function (PDF) to the half-plane, consider $\rho(\theta)$ as the probability density to be within an arbitrary range of angle θ. Let

δ be the smallest distance taken in the half-plane and $\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)$ as the current or initial positions. The next position in the half-plane, $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right)$, can be approximated by $\left(x^0_{\mathbb{H}_R} + \delta\cos(\theta), y^0_{\mathbb{H}_R} + \delta\sin(\theta)\right)$. We can write the angular PDF as,

$$\rho(\theta)\delta d\theta = \rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R} + \delta\cos(\theta), y^0_{\mathbb{H}_R} + \delta\sin(\theta)\right)\delta$$

By working locally, we can expand the right hand side to first order with respect to the half-plane positions, $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right)$.

$$\rho(\theta)\delta d\theta \approxeq \rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)\delta + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}}\right)\delta^2\cos(\theta) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right)\delta^2\sin(\theta)$$

Subtracting the initial probability density $\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)$ and dividing by δ gives,

$$\rho(\theta)d\theta - \rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right) \approxeq \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}}\right)\delta\cos(\theta) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right)\delta\sin(\theta)$$

Let $\Delta\rho_{\mathbb{H}_R}(\theta)d\theta = \rho(\theta)d\theta - \rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)$ be the localized angular probability density function. Then,

$$\Delta\rho_{\mathbb{H}_R}(\theta)d\theta \approxeq \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}}\right)\delta\cos(\theta) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right)\delta\sin(\theta)$$

Therefore, $\rho(\theta)d\theta = \Delta\rho_{\mathbb{H}_R}(\theta)d\theta + \rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)$, represents the localized angular probability density function. By normalization, with respect to the Euclidean metric, we achieve the final form of our sought out angular PDF. Let $\rho(\theta) = \rho_{\mathbb{H}_R}(\theta)$, then,

$$\rho_{\mathbb{H}_R}(\theta) = \frac{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}}\right)\delta\cos(\theta) + \left(\frac{\partial\rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right)\delta\sin(\theta)}{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)}$$

□

Now that we have a relationship of the angular PDF, $\left(\rho_{\mathbb{H}_R}(\theta)\right)$ from equation (2.54), we need the appropriate terms for the rate of change with respect to the half-plane points $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right)$. We start off from the uniform radial distribution function from [32],

derived in section §2.5.2.

$$\rho(r_{\mathbb{D}_R}) = \frac{\sinh\left(\frac{r_{\mathbb{D}_R}}{R}\right)}{R\left[\cosh\left(\frac{r_{\mathbb{D}_R}^{max}}{R}\right) - 1\right]} \sim \frac{1}{R} \cdot \exp\left(\frac{r_{\mathbb{D}_R}}{R}\right) \tag{2.55}$$

Where $r_{\mathbb{D}_R}$ is the Poincaré disk radial distance from the origin, equation (2.24). In [32], it was assumed that $r_{\mathbb{D}_R}^{max} >> R$ allowing the approximation in equation (2.55). From equation (2.24), we can represent equation (2.55) with respect to the Euclidean radius, $r_E \in [0, R)$.

$$\rho(r_{\mathbb{D}_R}) = \rho(r_E) = \frac{1}{R}\left(\frac{R + r_E}{R - r_E}\right) \tag{2.56}$$

We can represent the radius of the Poincaré disk with respect to the half-plane points, by a Mőbius transformation, from equation (2.36),

$$r_E = \sqrt{(x_{\mathbb{D}_R})^2 + (y_{\mathbb{D}_R})^2} = R\frac{\sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2Rx_{\mathbb{H}_R}\right)^2}}{\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2} \tag{2.57}$$

From here, we can represent equation (2.56) as a function dependent on points from the half-plane, $\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right)$, by $\rho(r_E) = \rho_{\mathbb{H}_R}\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right)$ from the new relation of r_E in equation (2.57). This will have the following form,

$$\rho_{\mathbb{H}_R}\left(x_{\mathbb{H}_R}, y_{\mathbb{H}_R}\right) = \frac{1}{R} \cdot \left[\tag{2.58a}$$

$$\frac{\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 + \sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2Rx_{\mathbb{H}_R}\right)^2}}{\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 - \sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2Rx_{\mathbb{H}_R}\right)^2}}\right] \tag{2.58b}$$

$$\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right) = \frac{1}{R} \cdot \left[\tag{2.58c}$$

$$\frac{\left(x^0_{\mathbb{H}_R}\right)^2 + \left(y^0_{\mathbb{H}_R} + R\right)^2 + \sqrt{\left(R^2 - (x^0_{\mathbb{H}_R})^2 - (y^0_{\mathbb{H}_R})^2\right)^2 + \left(2Rx^0_{\mathbb{H}_R}\right)^2}}{\left(x^0_{\mathbb{H}_R}\right)^2 + \left(y^0_{\mathbb{H}_R} + R\right)^2 - \sqrt{\left(R^2 - (x^0_{\mathbb{H}_R})^2 - (y^0_{\mathbb{H}_R})^2\right)^2 + \left(2Rx^0_{\mathbb{H}_R}\right)^2}}\right] \tag{2.58d}$$

The rate at which the radial PDF, equation (2.56), changes with respect to the Euclidean radius, r_E, is given by,

$$\frac{d\rho}{dr_E} = \frac{2}{\left(R - r_E\right)^2} \tag{2.59}$$

Computing the derivative of equation (2.57) with respect to $x_{\mathbb{H}_R}$ and $y_{\mathbb{H}_R}$, we get the following,

$$\frac{\partial r_E}{\partial x_{\mathbb{H}_R}} = \frac{4R^2 x_{\mathbb{H}_R} y_{\mathbb{H}_R}}{\left(\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2\right)\sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2}} \tag{2.60a}$$

$$\frac{\partial r_E}{\partial y_{\mathbb{H}_R}} = \frac{-2R^2\left[\left(y_{\mathbb{H}_R} + R\right)\left(2R\left(x_{\mathbb{H}_R}\right)^2 + \left(R - y_{\mathbb{H}_R}\right)\left(R + y_{\mathbb{H}_R}\right)^2\right) + \left(x_{\mathbb{H}_R}\right)^4\right]}{\left(\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2\right)^2\sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2}} \tag{2.60b}$$

To get rate of change with respect to the half-plane coordinates, we use the following relations,

$$\frac{\partial \rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}} = \frac{\partial \rho_{\mathbb{H}_R}}{\partial r_E}\frac{\partial r_E}{\partial x_{\mathbb{H}_R}} \tag{2.61a}$$

$$\frac{\partial \rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}} = \frac{\partial \rho_{\mathbb{H}_R}}{\partial r_E}\frac{\partial r_E}{\partial y_{\mathbb{H}_R}} \tag{2.61b}$$

After substituting equation (2.60) into equation (2.61) and working out two lines

of algebra, we get the final form,

$$\frac{\partial \rho_{\mathbb{H}_R}}{\partial x_{\mathbb{H}_R}} = \frac{8 x_{\mathbb{H}_R} y_{\mathbb{H}_R} \left(\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 \right)}{\sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2}} \cdot \left[\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 - \sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2} \right]^2 \tag{2.62a}$$

$$\frac{\partial \rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}} = \frac{-4 \left[\left(y_{\mathbb{H}_R} + R\right) \left(2R \left(x_{\mathbb{H}_R}\right)^2 + \left(R - y_{\mathbb{H}_R}\right) \left(R + y_{\mathbb{H}_R}\right)^2 \right) + \left(x_{\mathbb{H}_R}\right)^4 \right]}{\sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2}} \cdot \left[\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 - \sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2} \right]^2 \tag{2.62b}$$

By taking the limit $r_E \to R$ in the Poincaré disk metric, equation (2.23b), it is reasonable to assume $y_{\mathbb{H}_R} \to 0$ in the half-plane metric, equation (2.33). Thus, both metrics exhibit similar behavior. This allows us to drop the rate of change dependence on $x_{\mathbb{H}_R}$ in equation (2.54). This gives use the final approximated form as follows,

$$\rho_{\mathbb{H}_R}(\theta) = \frac{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right) + \left(\frac{\partial \rho_{\mathbb{H}_R}}{\partial y_{\mathbb{H}_R}}\right) \delta \sin(\theta)}{\rho_{\mathbb{H}_R}\left(x^0_{\mathbb{H}_R}, y^0_{\mathbb{H}_R}\right)} \tag{2.63}$$

The final version of equation (2.63) after normalization with Euclidean metric becomes,

$$\rho_{\mathbb{H}_R}(\theta) = \frac{1}{2\pi} \left[1 - \delta \sin(\theta) \cdot \frac{\left[\left(y_{\mathbb{H}_R} + R\right) \left(2R \left(x_{\mathbb{H}_R}\right)^2 + \left(R - y_{\mathbb{H}_R}\right) \left(R + y_{\mathbb{H}_R}\right)^2 \right) + \left(x_{\mathbb{H}_R}\right)^4 \right]}{y_{\mathbb{H}_R} \left(\left(x_{\mathbb{H}_R}\right)^2 + \left(y_{\mathbb{H}_R} + R\right)^2 \right) \sqrt{\left(R^2 - (x_{\mathbb{H}_R})^2 - (y_{\mathbb{H}_R})^2\right)^2 + \left(2R x_{\mathbb{H}_R}\right)^2}} \right] \tag{2.64}$$

By taking the inverse of equation (2.64), we have our $\theta(\rho_{\mathbb{H}_R})$ relation.

$$\theta(\rho_{\mathbb{H}_R}) = \arcsin\Bigg[\left(1-2\pi\rho_{\mathbb{H}_R}\right)\cdot \frac{y_{\mathbb{H}_R}\left(\left(x_{\mathbb{H}_R}\right)^2+\left(y_{\mathbb{H}_R}+R\right)^2\right)\sqrt{\left(R^2-(x_{\mathbb{H}_R})^2-(y_{\mathbb{H}_R})^2\right)^2+\left(2Rx_{\mathbb{H}_R}\right)^2}}{\delta\left[\left(y_{\mathbb{H}_R}+R\right)\left(2R\left(x_{\mathbb{H}_R}\right)^2+\left(R-y_{\mathbb{H}_R}\right)\left(R+y_{\mathbb{H}_R}\right)^2\right)+\left(x_{\mathbb{H}_R}\right)^4\right]}\Bigg] \tag{2.65}$$

From analyzing equation (2.64), in the limit of small δ values, $\rho_{\mathbb{H}_R}(\theta)$ approaches a constant value, $\left(\frac{1}{2\pi}\right)$. This implies, locally our angular probability density function is constant for small enough distance. Therefore, for small δ values, we have the uniform angular probability density function as,

$$\rho_{\mathbb{H}_R}(\theta) = \frac{1}{2\pi} \tag{2.66}$$

The probability of finding θ in the half-plane from our approximation and equation (2.66) comes out as,

$$\begin{aligned}\Pr(\theta>0) &= \int_0^{\theta}\rho_{\mathbb{H}_R}(\theta')\,d\theta' \\ &= \frac{1}{2\pi}\cdot\theta\end{aligned} \tag{2.67}$$

Note, we integrated with respect to θ only because the radial term was already considered in the proof of equation (2.54). By taking the inverse of equation (2.67), we arrive to the simulation θ as a function of $\Pr \in [0,1]$.

$$\theta(\Pr) = 2\pi\cdot\Pr \tag{2.68}$$

Simulation Localized Half-Plane Semi-Circle Center

Here, we show our approach for choosing the semi-circle center, x_c from equation (2.52), in the half-plane by assuming, locally, that the space is Euclidean, as discussed in section §2.5.2. By taking the smallest possible step, Δs, we have our tangent vector

at the localized point as,

$$\vec{r_T} = \begin{Bmatrix} x_T \\ y_T \end{Bmatrix} = \begin{Bmatrix} \Delta s \cdot \cos(\theta) + x_1 \\ \Delta s \cdot \sin(\theta) + y_1 \end{Bmatrix} \tag{2.69}$$

Our corresponding tangent slope is then,

$$m_T = \frac{y_T - y_1}{x_T - x_1} = \tan(\theta) \tag{2.70}$$

We get the following linear equation from the tangent slope,

$$y_T = m_T \cdot (x - x_1) + y_1 = \tan(\theta) \cdot (x - x_1) + y_1 \tag{2.71}$$

Next, we construct the normal line equation to our tangent relation. The normal to the tangent vector equation (2.69) is,

$$\vec{r_N} = \begin{Bmatrix} x_N \\ y_N \end{Bmatrix} = \begin{Bmatrix} \Delta s \cdot \cos(\theta + \frac{\pi}{2}) + x_1 \\ \Delta s \cdot \sin(\theta + \frac{\pi}{2}) + y_1 \end{Bmatrix} = \begin{Bmatrix} -\Delta s \cdot \sin(\theta) + x_1 \\ \Delta s \cdot \cos(\theta) + y_1 \end{Bmatrix} \tag{2.72}$$

Our corresponding normal slope from the equation (2.72) is,

$$m_N = \frac{y_N - y_1}{x_N - x_1} = -\cot(\theta) \tag{2.73}$$

We get the following linear equation from the normal slope,

$$y_N = m_T \cdot (x - x_1) + y_1 = -\cot(\theta) \cdot (x - x_1) + y_1 \tag{2.74}$$

Note, $m_N = -\frac{1}{m_T}$. To find our semi-circle center x_c for our simulation, we require $y_N = 0$ and $x = x_c$ in equation (2.74). We have the following form for our semi-circle center x_c,

$$x_c = -\frac{y_1}{m_N} + x_1 = m_T y_1 + x_1 = \tan(\theta) y_1 + x_1 \tag{2.75}$$

The angle θ is chosen from a uniform probability distribution, discussed in section §2.5.2, equation (2.68).

Poincaré Disk Radial Distribution

Here, we derive the radial probability distribution for the Poincaré disk, by working with its metric, equation (2.23). We assume a uniform radial distribution in this map. We consider the entire Poincaré disk area from equation (2.28), taken at some arbitrary max Poincaré radial distance from the origin and label it r_p^{max}. Then divide the circumference, equation (2.27), which is a function of $r_{\mathbb{D}_R} \in \left[0, r_p^{max}\right]$, by the maximum area yields the uniform radial probability density function.

$$\rho(r_{\mathbb{D}_R}) = \frac{C_{\mathbb{D}_R}(r_{\mathbb{D}_R})}{A_{\mathbb{D}_R}(r_{\mathbb{D}_R} = r_p^{max})} = \frac{\sinh\left(\frac{r_{\mathbb{D}_R}}{R}\right)}{R\left[\cosh\left(\frac{r_{\mathbb{D}_R}=r_p^{max}}{R}\right) - 1\right]} \tag{2.76}$$

The probability of being at an arbitrary Poincaré radial distance, $\left(r_{\mathbb{D}_R}\right)$, is shown in equation (2.77). Note, the derivative of the radial distance is $\frac{dr_{\mathbb{D}_R}}{dr} = dr^{'} = \frac{2R^2}{R^2-r^2}dr$, from equation (2.24), where $r = r_E$ and $r_E = R\tanh\left(\frac{r_{\mathbb{D}_R}}{2R}\right)$.

$$\begin{aligned}
\Pr(r_{\mathbb{D}_R} > 0) &= \int_{r_{\mathbb{D}_R}}^{r_p^{max}} \rho(r_{\mathbb{D}_R}) \frac{2R^2}{R^2 - r^2}\, dr \\
&= \int_{r_{\mathbb{D}_R}}^{r_p^{max}} \frac{\sinh\left(\frac{r^{'}}{R}\right)}{R\left[\cosh\left(\frac{r_p^{max}}{R}\right) - 1\right]}\, dr^{'} \\
&= \frac{\left[\cosh\left(\frac{r_p^{max}}{R}\right) - \cosh\left(\frac{r_{\mathbb{D}_R}}{R}\right)\right]}{\cosh\left(\frac{r_p^{max}}{R}\right) - 1}
\end{aligned} \tag{2.77}$$

By taking the inverse of equation (2.77) and solving for $r_{\mathbb{D}_R}$, we get the Poincaré disk radial distance as a function of the probability, $\Pr \in (0, 1)$.

$$r_{\mathbb{D}_R}(Pr) = R\,\text{arcCosh}\left[\cosh\left(\frac{r_p^{max}}{R}\right) - \Pr\cdot(\cosh\left(\frac{r_p^{max}}{R}\right) - 1)\right] \tag{2.78}$$

For a random walk on the Poincaré disk, the radial length can be drawn from equation (2.78) with $\Pr(r_{\mathbb{D}_R} > 0) \in [0, 1]$. In the case of Euclidean space we have the following relations: $\rho(r) = \frac{2r}{R^2-r_{min}^2}$, $\Pr(r < R) = \frac{r^2-r_{min}^2}{R^2-r_{min}^2}$, $r(\Pr) = \sqrt{r_{min}^2 + \Pr\left(R^2 - r_{min}^2\right)}$, where r_{min} is the minimum Euclidean disk radius. For simplicity, we let $r_{min} = 0$ for

simulations and $\Pr(r < R) \in [0, 1]$.

2.5.3 Spherical Geometry

Here, we present the position equation, geodesic and central angle relationship of the sphere along with quaternions, the iterative methodology of updating a position on a sphere, found in references [39, 46].

Position, Geodesic and Central Angle

The **sphere** of radius R is defined as,

$$\mathbb{S}_R^2 \in \left\{(x, y, z) \in \mathbb{R} \mid x^2 + y^2 + z^2 = R^2\right\} \tag{2.79}$$

With the following parameterized form,

$$x(\theta, \phi) = R\cos(\theta)\cos(\phi) \tag{2.80a}$$
$$y(\theta, \phi) = R\cos(\theta)\sin(\phi) \tag{2.80b}$$
$$z(\theta, \phi) = R\sin(\theta) \tag{2.80c}$$

where $\theta \in \left[-\pi/2, \pi/2\right]$ and $\phi \in [0, 2\pi]$. The origin is located on the x-axis with $(\theta, \phi) = (0, 0)$, and $(x, y, z) = (R, 0, 0)$. The geodesic is equal to $\Delta s = R \cdot \Delta\sigma_c$, where σ_c is the central angle, given by the spherical law of cosines and has the following form,

$$\Delta\sigma_c = \arccos\left[\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)\cos(\phi_2 - \phi_1)\right] \tag{2.81}$$

Given a geodesic distance Δs between two points on a sphere of radius R, the central angle will be $\Delta\sigma_c = \frac{\Delta s}{R}$.

$$\Delta s = R \cdot \arccos\left[\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)\cos(\phi_2 - \phi_1)\right] \tag{2.82}$$

Quaternion

A rigorous explanation of quaternion can be found in reference [46]. The idea is to extend our vector $\vec{u} = [x, y, z]$ in $\mathbb{R}^3$ to the extended version of complex space represented as $u_{ex} = [0, x, y, z]$ or $u_{ex} = 0 + x\hat{i} + y\hat{j} + z\hat{k}$. Quaternion have the multiplication property of $\hat{i}^2 = \hat{j}^2 = \hat{k}^2 = \hat{i}\hat{j}\hat{k} = -1$.

In $\mathbb{R}^3$, a rotation of angle $\Delta\sigma_{central}$ is performed about a unit vector $\hat{v}$ of the following form,

$$\hat{v} = [a, b, c] \tag{2.83}$$

In the extended complex plane, this is a multiplication with the quaternion, $q = \left[\cos\left(\frac{\Delta\sigma_c}{2}\right), \sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot \hat{v}\right]$. The written out version is,

$$q\left(\Delta\sigma_c\right) = \begin{pmatrix} \cos\left(\frac{\Delta\sigma_c}{2}\right) \\ \sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot a \\ \sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot b \\ \sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot c \end{pmatrix} \tag{2.84}$$

The inverse is given as the complex conjugate of equation (2.84),

$$q^{-1}\left(\Delta\sigma_c\right) = \begin{pmatrix} \cos\left(\frac{\Delta\sigma_c}{2}\right) \\ -\sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot a \\ -\sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot b \\ -\sin\left(\frac{\Delta\sigma_c}{2}\right) \cdot c \end{pmatrix} \tag{2.85}$$

with the property of $q \cdot q^{-1} = 1$. Given the location of a point on the sphere $\vec{u}$, the new position after a rotation of angle $\Delta\sigma_c$ about a unit vector $\hat{v}$, is given by $u_{ex}^{'} = q \cdot u_{ex} \cdot q^{-1} = Q \cdot q^{-1} = \left[g, x^{'}, y^{'}, z^{'}\right]$, where $Q = q \cdot u_{ex} = [L, m, n, p]$.

$$Q = [L, m, n, p] = \begin{pmatrix} -\sin\left(\frac{\Delta\sigma_c}{2}\right)\left(a \cdot x + b \cdot y + c \cdot z\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot x + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(b \cdot z - c \cdot y\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot y + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(c \cdot x - a \cdot z\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot z + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(a \cdot y - b \cdot x\right) \end{pmatrix} \tag{2.86}$$

The final multiplication is $u_{ex}^{'} = Q \cdot q^{-1} = \left[g, x^{'}, y^{'}, z^{'}\right]$.

$$u_{ex}^{'} = \left[g, x^{'}, y^{'}, z^{'}\right] = \begin{pmatrix} \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot L + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(a \cdot m + b \cdot n + c \cdot p\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot m + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(b \cdot p - a \cdot L - c \cdot n\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot n + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(c \cdot m - b \cdot L - a \cdot p\right) \\ \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot p + \sin\left(\frac{\Delta\sigma_c}{2}\right)\left(a \cdot n - c \cdot L - b \cdot m\right) \end{pmatrix} \tag{2.87}$$

Therefore, our new position on the sphere, with a rotation of $\Delta\sigma_c$ about the unit vector $\hat{v} = [a, b, c]$ from the point $\vec{u} = [x, y, z]$ is,

$$x^{'} = \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot m + \sin\left(\frac{\Delta\sigma_c}{2}\right)(b \cdot p - a \cdot L - c \cdot n) \tag{2.88a}$$

$$y^{'} = \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot n + \sin\left(\frac{\Delta\sigma_c}{2}\right)(c \cdot m - b \cdot L - a \cdot p) \tag{2.88b}$$

$$z^{'} = \cos\left(\frac{\Delta\sigma_c}{2}\right) \cdot p + \sin\left(\frac{\Delta\sigma_c}{2}\right)(a \cdot n - c \cdot L - b \cdot m) \tag{2.88c}$$

The unit vector $\hat{v}$ is computed by randomly choosing new angles, (θ, ϕ), producing a new position vector $\vec{u_2}$. We then compute the cross product from our initial location of our geodesic path $\vec{u}$, $\vec{v} = \vec{u} \times \vec{u_2}$. The unit vector will then be $\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$.

2.5.4 Simulation

This section will prove the Lévy distributions used for our random walk, present the random walk algorithm for each space, the distance equations used in computing the MSD for our simulated random walks and show the various MSD theoretical models that we fit.

Lévy Distribution

Lévy distributions have two cases to consider. First, when the upper bound on the distribution is finite, second when it is infinite. Physically, this would be considered when the space is bounded or unbounded.

- Case 1: $\boxed{\Delta s_{max} = \text{Finite}}$

For Lévy distribution, we have as the normalized probability density function with respect to a lower and upper bound as follows,

$$\rho(\Delta s) = \frac{\mu - 1}{\Delta s_0^{1-\mu} - \Delta s_{max}^{1-\mu}} (\Delta s)^{-\mu} \, \delta\left(|\Delta s| - vt\right) \tag{2.89}$$

where Δs_0 is the smallest geodesic length, Δs_{max} is the largest geodesic length and μ is the Lévy exponent such that $\mu \in (1, 3)$. We apply the Dirac delta function, $\delta(|\Delta s| - vt)$, to ensure we have a ballistic relation constrained by a constant velocity v. This couples time and length together allowing for Lévy walk to occur.

The probability of a geodesic length, Δs, is given by the integration of equation (2.89), $P(\Delta s > \Delta s_0)$, from Δs to Δs_{max}.

$$\Pr(\Delta s > \Delta s_0) = \frac{1}{\Delta s_0^{1-\mu} - \Delta s_{max}^{1-\mu}} \left(\Delta s^{1-\mu} - \Delta s_{max}^{1-\mu}\right) \tag{2.90}$$

where $\Pr(\Delta s > \Delta s_0) \in [0, 1]$. Taking the inverse of equation (2.90) and solving for Δs, we find the geodesic path length as,

$$\Delta s(\Pr) = \left[\Delta s_{max}^{1-\mu} + \Pr\left(\Delta s_0^{1-\mu} - \Delta s_{max}^{1-\mu}\right)\right]^{\frac{1}{1-\mu}} \tag{2.91}$$

• Case 2: $\boxed{\Delta s_{max} \to \infty}$

In the case of an unrestricted upper bound, we have the normalized probability density function with a lower bound and in the large limit as,

$$\rho(\Delta s) \approx \frac{\mu - 1}{\Delta s_0} \left(\frac{\Delta s}{\Delta s_0}\right)^{-\mu} \delta\left(|\Delta s| - vt\right) \tag{2.92}$$

where Δs_0 is the smallest geodesic path length the walker can take and μ is the Lévy exponent. Again, we apply the Dirac delta function, $\delta(|\Delta s| - vt)$, to ensure we have a ballistic relation constrained by a constant velocity v. The probability of a geodesic length, Δs, in this case is given by integrating equation (2.92), $P(\Delta s > \Delta s_0)$, from Δs to ∞.

$$\Pr(\Delta s > \Delta s_0) \approx \left(\frac{\Delta s}{\Delta s_0}\right)^{-\mu+1} \tag{2.93}$$

Taking the inverse of equation (2.93) and solving for Δs, we find the geodesic path length as,

$$\Delta s(\Pr) \approx \Delta s_0 \times (\Pr)^{\frac{1}{1-\mu}} \tag{2.94}$$

Simulation Random Walk Algorithm

We show the generalized position equations for Euclidean, hyperbolic half-plane model and spherical space. The half-plane equations proof is shown in section §2.5.2 and for the sphere in §2.5.3. For Euclidean, geodesic paths are straight lines with position

equations,

$$x_{i+1}(\delta,\theta) = x_i + \delta \cdot \cos(\theta) \tag{2.95a}$$

$$y_{i+1}(\delta,\theta) = y_i + \delta \cdot \sin(\theta) \tag{2.95b}$$

In choosing the Euclidean geodesic path length, Δs, we break the path into n fixed step lengths, δ, where $\Delta s = \sum^n \delta$. For hyperbolic space, using the half-plane model, geodesics are semi-circles. In order to implement a random walk in this space, we formulated a local approximation to pick, randomly, the semi-circle center, shown in section §2.5.2. The position equations, in the hyperbolic half-plane model and derived in section §2.5.2, are,

$$x_c = \tan(\theta) \cdot y_i + x_i \tag{2.96a}$$

$$r = \frac{R}{C} = \sqrt{(x_i - x_c)^2 + (y_i)^2} \tag{2.96b}$$

$$s_1(y_i) = -R \ln\left(\frac{1}{y_i}\left(r + \sqrt{(r)^2 - (y_i)^2}\right)\right) \tag{2.96c}$$

$$x_{i+1}(\delta) = x_c + r \tanh\left(\frac{s_1 + \delta}{R}\right) \tag{2.96d}$$

$$y_{i+1}(\delta) = r \operatorname{sech}\left(\frac{s_1 + \delta}{R}\right) \tag{2.96e}$$

For the hyperbolic geodesic path length, Δs, each step is the sum of everything prior for the geodesic, such that $\Delta s = \sum_1^m \delta$, where $m \in [1, n]$. Both Euclidean and hyperbolic use geodesic equation (2.94) with polar angles drawn form a uniform distribution, $\theta = 2\pi \cdot \Pr$.

In spherical space, geodesics are arcs around the sphere. By using the method of quaternions, discussed in section §2.5.3, we get the following position equations,

$$x_{i+1}(\delta) = \cos\left(\frac{\sigma_c}{2}\right) \cdot m + \sin\left(\frac{\sigma_c}{2}\right)(b \cdot p - a \cdot L - c \cdot n) \tag{2.97a}$$

$$y_{i+1}(\delta) = \cos\left(\frac{\sigma_c}{2}\right) \cdot n + \sin\left(\frac{\sigma_c}{2}\right)(c \cdot m - b \cdot L - a \cdot p) \tag{2.97b}$$

$$z_{i+1}(\delta) = \cos\left(\frac{\sigma_c}{2}\right) \cdot p + \sin\left(\frac{\sigma_c}{2}\right)(a \cdot n - c \cdot L - b \cdot m) \tag{2.97c}$$

Spherical space use geodesic equation (2.91), where the geodesic path length, Δs, is

broken into n fixed step lengths, δ, where $\Delta s = \sum^{n} \delta$. This then gives the incremental central angle $\sigma_c = \frac{\delta}{R}$, used as the amount rotated about a randomly chosen unit vector. As the walker traverses the path, our unit vector remains unchanged.

Simulation MSD Calculation

The simulation MSD is computed by performing a sliding window average. In the case of Euclidean, the MSD is given by equation (2.98). For the sphere, equation (2.99). For hyperboloid, using the half-plane model, equation (2.100). Let T represent the max simulation time, then $\Delta t \in [1, T-1]$ and $i \in [1, T - \Delta t]$.

Euclidean sliding window MSD is,

$$\Delta s^2(\Delta t) = \big(x(i+\Delta t) - x(i)\big)^2 + \big(y(i+\Delta t) - y(i)\big)^2 \tag{2.98}$$

For position on the sphere, the corresponding MSD is,

$$\Delta s^2(\Delta t) = \left[2R \cdot \arcsin\left(\frac{\sqrt{\big(x(i+\Delta t) - x(i)\big)^2 + \big(y(i+\Delta t) - y(i)\big)^2 + \big(z(i+\Delta t) - z(i)\big)^2}}{2 \cdot R}\right)\right]^2 \tag{2.99}$$

For position in the half-plane model, the corresponding MSD is,

$$\Delta s^2(\Delta t) = \left[R \cdot \operatorname{arcCosh}\left(\frac{\big(x(i+\Delta t) - x(i)\big)^2 + y(i+\Delta t)^2 + y(i)^2}{2 \cdot y(i) \cdot y(i+\Delta t)}\right)\right]^2 \tag{2.100}$$

Fitted Mean-Squared Displacement

Our analysis consists of modeling eight MSD forms, shown in equation (2.101). The notations of the MSD model is, $\langle \Delta s^2(t) \rangle_{i,j}$, where i represents the polynomial order and j as the number of parameters being fit. The negative sign, $(-)$, in the second

order term corresponds to spherical space and positive, (+), for hyperbolic.

$$\langle \Delta s^2(t) \rangle_{1,1} = 4\widetilde{D}t^a \tag{2.101a}$$

$$\langle \Delta s^2(t) \rangle_{1,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \tag{2.101b}$$

$$\langle \Delta s^2(t) \rangle_{2,1} = 4\widetilde{D}t^a \pm \frac{4}{3R^2}\left(\widetilde{D}t^a\right)^2 \tag{2.101c}$$

$$\langle \Delta s^2(t) \rangle_{2,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \frac{4}{3}R^2\left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^2 \tag{2.101d}$$

$$\langle \Delta s^2(t) \rangle_{2,3} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \widetilde{\beta}_1 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^2 \tag{2.101e}$$

$$\langle \Delta s^2(t) \rangle_{3,1} = 4\widetilde{D}t^a \pm \frac{4}{3R^2}\left(\widetilde{D}t^a\right)^2 - \frac{8}{45R^4}\left(\widetilde{D}t^a\right)^3 \tag{2.101f}$$

$$\langle \Delta s^2(t) \rangle_{3,2} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \frac{4}{3}R^2\left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^2 - \frac{8}{45}R^2\left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^3 \tag{2.101g}$$

$$\langle \Delta s^2(t) \rangle_{3,4} = 4\widetilde{D}t^{\widetilde{\alpha}} \pm \widetilde{\beta}_1 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^2 - \widetilde{\beta}_2 \cdot \left(\frac{\widetilde{D}t^{\widetilde{\alpha}}}{R^2}\right)^3 \tag{2.101h}$$

The anomalous exponent a will always have the range shown in reference [13], depending on the Lévy exponent μ, and is considered a known, fixed value in the fit. Greek letters, $\widetilde{\alpha}, \widetilde{\beta}_1, \widetilde{\beta}_2$, along with the diffusion constant $\widetilde{D}$, are parameters that will be fitted. The case of fitting only $\widetilde{D}, \widetilde{\beta}_1, \widetilde{\beta}_2$ with the known exponent a fixed, did not produce stable and consistent results and is excluded from our MSD forms.

Chapter 3

Intracellular Transport on Dynamic Actin Networks

3.1 Introduction

The internal dynamics of eukaryotic (animal) cells are very rich, and complex. The environment of the inner cell (the cytosol/cytoplasm) consists of a multitude of molecular components that can work in tandem, or completely separate along very complicated molecular pathways, which later dictate cell behavior and function downstream. Moreover, cellular functions cover a wide dynamic range of spatiotemporal scales, which makes describing the cell mathematically overall extremely challenging. One major operational component of cellular function is its internal cytoskeleton. This skeleton can give rise to bulk mechanical properties of the cell [3, 17]. However, it is better described as a dynamic network that constantly changes its topology and connectivity, and it is not static. The dynamic aspect of the network can cause regions to become crowded in some parts and become unpopulated in other parts. Not only does this network change rapidly over time, but also acts as the roadway for molecular cargo transport within the cell. It has been shown experimentally that cargo transport is strongly influenced by the dynamics in the network geometry [7].

In most eukaryotic cells, intracellular transport of cargos is a vital mechanism for cellular functions such as in the transportation of vesicles containing proteins like insulin [47], or transportation of organelles like mitochondria within neurons [48] to various targets throughout the cell. For nanometer scale molecules such as bacteria,

over small distances ($< \mu m$), transport by diffusion is the sufficient. However, diffusion becomes slow for larger cargos over large distances, requiring an additional mode of transport by the cytoskeletal network. Two types of transport phases can occur. One is active transport or ballistic motion, where ATP-powered molecular motors, such as kinesin, dynein, and myosin walk in a hand-over-hand style of motion along the cell's cytoskeletal network (consist of microtubules and actin filaments) while carrying cargo [16]. Another phase is passive transport or diffusive motion, where cargo diffuses throughout the cytoplasm until it encounters a filament where binding and unbinding can occur. In systems with static network, it has been shown that the topology of the network influences transport time, and filaments introduce superdiffusive behavior in cargos at short times [17]. However, in a more realistic system, networks vary with time.

In this chapter, we expand a previous approach that not only captures many of the critical features of intracellular transport of cargo, but also allows theoretical investigation of optimal transport for dynamic cytoskeletal network, specifically for actin filaments. The objective is to understand how cargo transportation is affected when the network of filaments become time-dependent. This allows us to answer, how do the dynamics affect cargo transport from one point to another in the cell? Can the dynamics be tuned to optimize transport? Actin filaments are known to be unstable [49]. These instabilities lead to dynamic networks. Polymerization can occur, which leads to growth at the plus end and depolymerization, which leads to shrinkage at the minus end. A balance of both can lead to steady-state treadmilling. Treadmilling speed can be tuned by the cell [49]. It has been shown that actin dynamics can regulate actin-based movement of membrane organelles [7]. To incorporate these behaviors, the parameters controlling transport in our simulation are actin filaments, which are up to a few microns in length, filament treadmilling speed ranging between $20 - 160\ \frac{nm}{s}$ [50, 51]. Cargos are carried along actin by myosin motors, and these motors have speeds between $60 - 200\ \frac{nm}{s}$ [51]. Cargo can attach and detach from filament with rates k_{on}, k_{off}.

3.2 Methods

Our work build on previous work [3, 17] where we start with a simple eukaryotic cell, represented by an annulus, with inner nucleus radius of 5 μm, outer cell membrane radius of 15 μm. Each cargo has a radius of 100 nm. For the simulation, cargos are represented as random walkers that alternate between passive transport within the cytoplasm, and active transport along an explicit cytoskeletal network. At the start of the walk, each cargo starts near the nucleus in the diffusive phase and move until they reach the cell's outer membrane (exocytosis) or maximum time. The nucleus has a boundary that will reflect the cargo. If it is near a filament, it can attach and switch to the ballistic phase. Our simulation did not allow switching to different filament if it encountered an intersection. During each time-step, actin filaments are modified such that it produces treadmilling motion. The positive end grows while the negative end shrinks, at the same speed. Also, if/when the filaments reach a minimum length, the cargo switches to the diffusive phase and the filament is rearranged inside the cell. We use physiological parameter values of diffusion, cargo speed, attachment and detachment rate, as well as the physical time-step from [3, 7, 17], to simulate our system, in order to better compare our results and justify our findings.

The actin filaments will have a range of growth/decay speed along with the number of filaments; whereas for the cargo, it will have a range of detachment rates which we will explore. The system parameters can be found in the Appendix §3.5, Table 3.1. Figure 3.1 shows the Monte Carlo simulation of our system with parameters used in previous work [3, 17]. This represents an actin network that treadmills, therefore, the networks are dynamical. In this case, the cargo attaches to the cyan filament, switching to ballistic motion. While the filament treadmills, the cargo still walks and detaches at some point, switching to diffusive motion. The brown lines are the cargos full trajectory in this example and the dashed lines are the past locations of the minus end of the filaments.

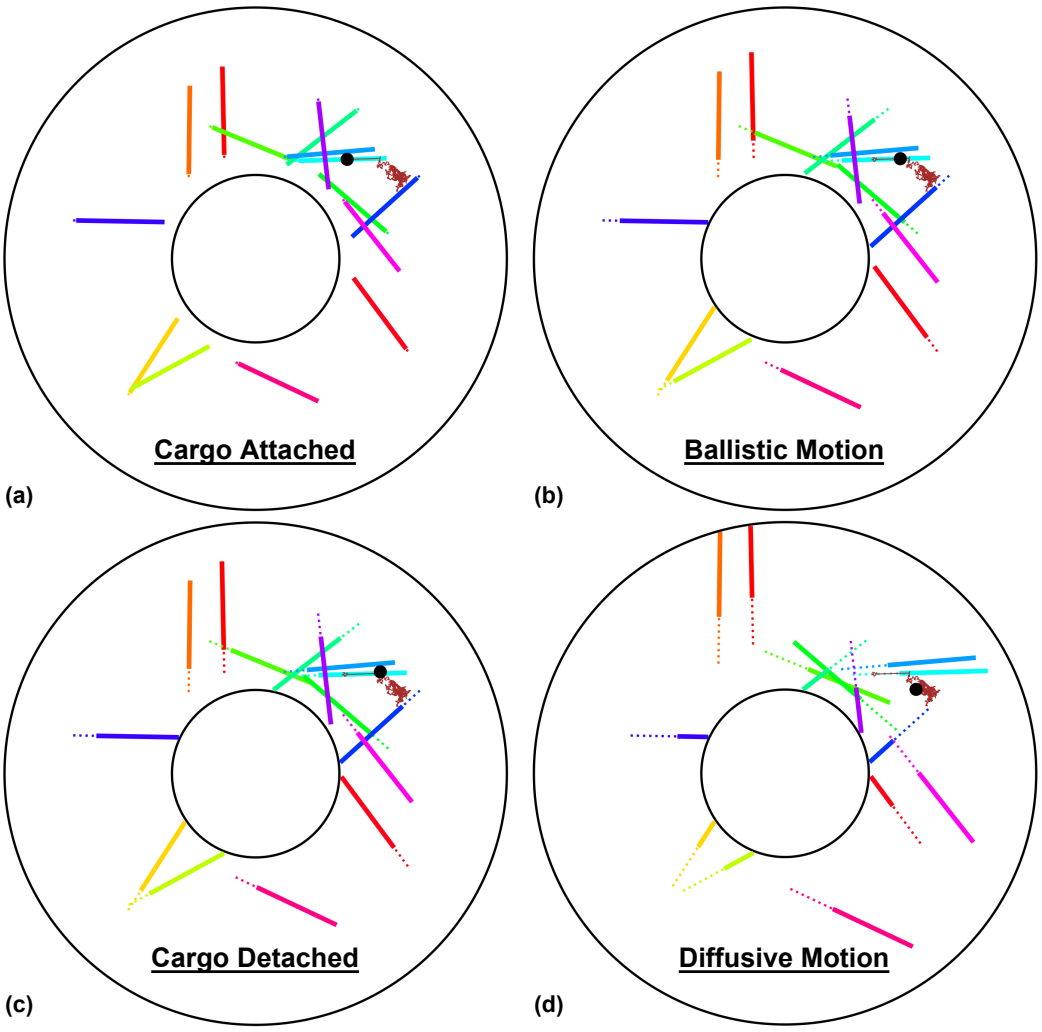

Figure 3.1: Simulation snapshot of our system with parameter values, $D = 0.051\ \frac{\mu m^2}{s}$, $a = 0.1\ \mu m$, $N_{fil} = 15$, $L = 5\ \mu m$, $k_{on} = 3.0\ s^{-1}$, $k_{off} = 0.5\ s^{-1}$, $v_c = 1.0\frac{\mu m}{s}$, $v_f = 0.6\frac{\mu m}{s}$. (a) Cargo attaches to the filament, $\big(t_{step} = 11\big)$. (b) Cargo moves ballistically on filament with speed v_c, $\big(t_{step} = 40\big)$. (c) Cargo detaches from filament, $\big(t_{step} = 52\big)$. (d) Cargo diffuses, $\big(t_{step} = 157\big)$.

3.3 Results

3.3.1 Enhanced MSD and Optimal MFPT

We start our analysis by observing how the cargo spreads throughout the cell from its starting position, near the nucleus, represented by the mean-squared displacement (MSD). Shown in Figure 3.2, in the larger plot, is the cargos MSD versus Time, for different actin filament treadmilling speeds v_f, ranging from ten-times less to

ten-times more of the cargos speed, $v_c = 0.0612\ \frac{\mu m}{s}$. The simulation is performed with the fixed parameters of 100 filaments each 6 μm in length, detachment rate of $k_{off} = 0.06\ s^{-1}$ and attachment $k_{on} = 5.0\ s^{-1}$. The cargo moves ballistically while on the actin network with speed v_c. We notice when the filament speed equals the cargo speed, i.e., $v_f = v_c$, a drastic increase in MSD occurs, shown by the cyan line, such that more of the cell is covered, which we label as our enhanced region, indicating that the distance traveled is not monotonic with filament treadmilling speed. The inset has plots of MSD versus Filament treadmilling speed at 800 s. This shows that even at longer times, the system still remains enhanced. Therefore, mean-squared displacement is enhanced for particular filament speeds. In this case, when filament speed equals the cargo speed. Given these findings, what does this mean for optimal transport?

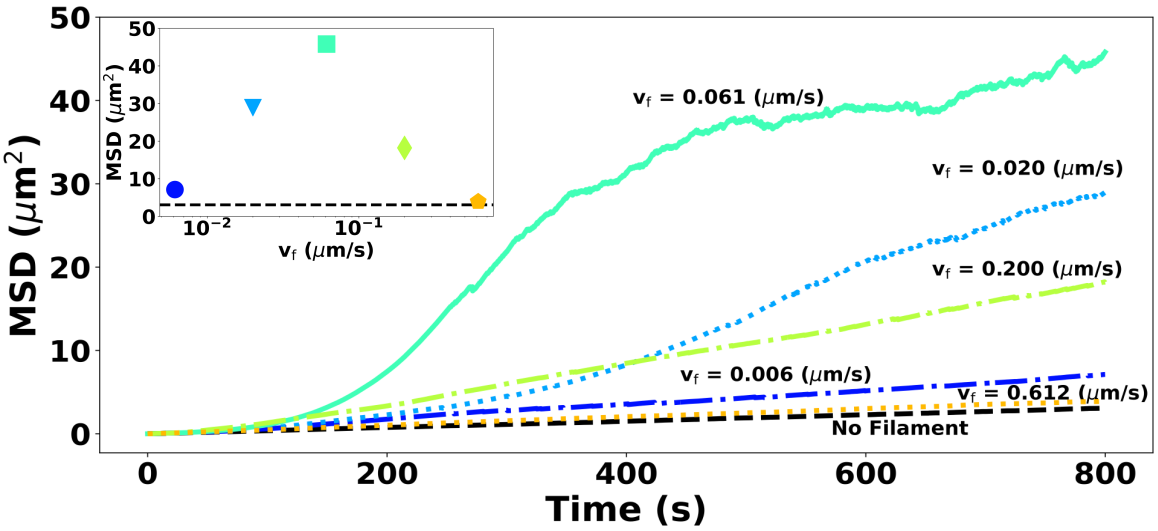

Figure 3.2: Cargo MSD vs time from our simulation for various filament speeds, shows at early time, for specific filament speeds, ($v_f \simeq v_c = 0.0612\ \frac{\mu m}{s}$), closed to the cargo speed enhanced MSD. Inset: MSD vs filament treadmilling speed at 800 s, shows the MSD enhancement persists even at late times.

To understand optimal transport, we look at the mean-first passage time (MFPT), which is defined as the first time the cargo reaches its target destination, the cell's membrane. Shown in Figure 3.3, for the larger plot is the cargos MFPT versus Filament treadmilling speed, blue squares. The results shown are for the same setup with 100 filaments each 6 μm in length. We notice that the MFPT drops then increases again. It attains the minimum value for a filament speed around the cargo

speed, $v_f = v_c$, which we label as optimal. To test the robustness of our system, a quadratic fit is performed on the data around this optimal speed, red-dashed line, which also results in a minima, red square, around our simulations optimal speed. This is also the point where the time spent on the filament is maximized, shown in the inset as blue circles. Interestingly, this minimum is in the physically relevant filament speed range for *in vivo* experiments, $(20 - 160\ \frac{nm}{s})$, yellow shaded region. This suggests that filament speeds can be tuned to optimize transport. Why is there a minimum at this value? Can we tune this value?

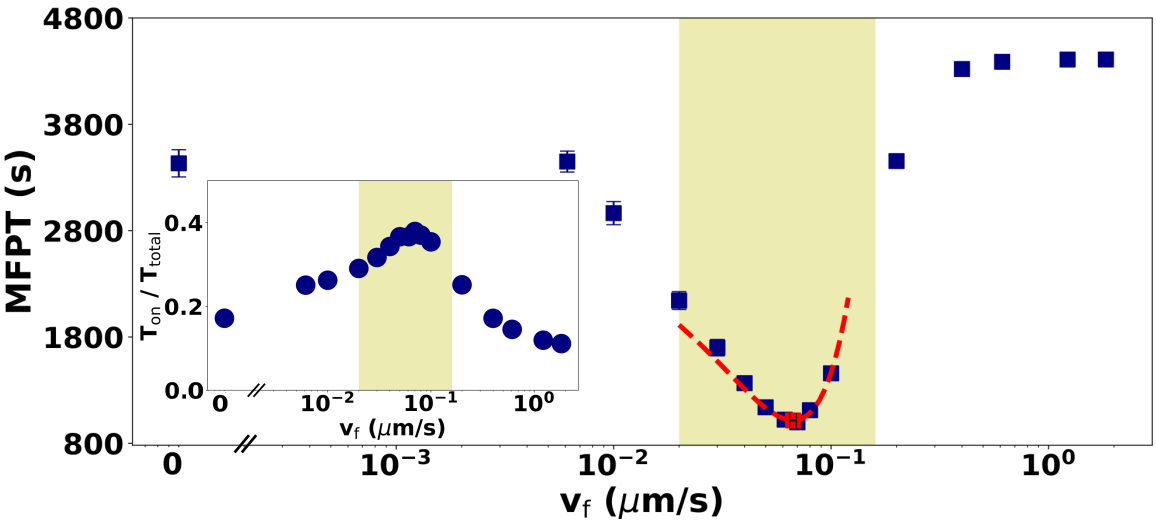

Figure 3.3: Cargo MFPT vs filament treadmilling speed from our simulation, shows a minima occurs for filament speed around the cargo speed. Inset: Fraction of time spent by cargo on filament vs filament speed, shows that, around the optimal speed, a maximum occurs.

3.3.2 Tuned Speed Range Dependencies

To understand how this minimum in MFPT arises, as seen in Figure 3.3, a closer observation is needed between the cargo and filament. The cargo can attach to a filament with attachment rate k_{on}; conversely it can detach with detachment rate k_{off}. Lets consider a single cargo walking on a filament in detail. Imagine walking on a track that shrinks from behind. Once in a while, you walk off the track; this time is defined as $\tau_b = \frac{1}{k_{off}}$, and you start to diffuse. While diffusing for some time, you reattach to the track, define this time as $\tau_u = \frac{1}{k_{on}}$. In this time, the track/filament ends move a distance, $d_f = v_f \tau_u = \frac{v_f}{k_{on}}$, with v_f being the filament treadmilling speed.

Relative to this filament, while the walker/cargo is attached, it moves a distance $d_c = \widetilde{v_c}\tau_b = \frac{v_c - v_f}{k_{off}}$, where v_c is the cargos ballistic speed. To maximize the time a cargo spends on the filament, as in the inset of Figure 3.3, we want the cargo to keep a constant distance from the ends of the filament. If we consider the criteria that the filament ends never reach the cargo, we can solve for the filament speed by equating both distance terms together, $d_f = d_c$. This gives the optimal filament speed, v_f^o, as a function of the cargo speed, cargo attachment and detachment rate, equation (3.1).

$$v_f^o = \frac{v_c}{1 + \frac{k_{off}}{k_{on}}} \tag{3.1}$$

Therefore, optimal actin transport of cargo requires $v_f = v_f^o$. Cargo never encounters the filament ends and remains on until it diffuses c_{rad} away from the filament. For example, if we consider $k_{off} \ll k_{on}$, transportation of cargo is optimized when actin filament co-move with cargo, $v_f = v_f^o = v_c$. Interestingly, equation (3.1) gives us an expression such that actin treadmilling speed can be tuned by cargo speed, attachment and detachment rate, (v_c, k_{on}, k_{off}).

To better understand the derived expression, shown in Figure 3.4a for our simulation, is the ratio of optimal filament speed to cargo speed versus the ratio of cargo detachment to attachment rates. The points are derived from finding the minima in the MFPT for different filament densities, where the red-dashed line is the predicted tuning equation (3.1). We see an excellent agreement between the simulation and prediction. From the expression, the optimal filament speed can be tuned by the cargoes attachment and detachment rates. When attachment rate, k_{on}, is large compared to the detachment rate, k_{off}, the cargo is basically always on the filament. For the cargo to not reach the filament ends, the filament must co-move with the cargo. This requires the filament speed to move closer to the cargo speed. From the work shown earlier, the filament speed, v_f, is approximately the cargos speed, v_c. When the detachment rate, k_{off}, increases, the optimal speed will fall. In the case when the detachment to attachment ratio, $\frac{k_{off}}{k_{on}}$, is close to 1, the cargo spends as much time off the filament as it does on. In other words, the chances of cargo to attach to the filament is the same as detaching. The filament should treadmill slow enough such that the ends do not reach the cargo, therefore the optimal speed will decrease. For optimal transport, the cell can use this as a control knob to tune the filaments speed.

Thus, controlling where optimal is.

The work shown so far are for detachment rates less than or equal to the attachment rate, $k_{off} \leq k_{on}$. If we consider the case when cargo detachment rate is larger than attachment, $k_{off} > k_{on}$, our optimal filament speed relationship, (3.1), does not hold anymore. Shown in Figure 3.4b are results for the ratio of filament to cargo speed versus number of filament. The simulation has detachment rate $k_{off} = 10\ s^{-1}$, attachments rate $k_{on} = 5\ s^{-1}$ and filament lengths $L = 2, 6\ \mu m$ (light blue, orange). Unlike the results in Figure 3.4a, we observe there exist a dependence on the network density for optimal transport. In addition, the ratio of time spent on the filament, shown in the inset for 100 filaments each 6 μm as blue circles, has a monotonic behavior as we increase filament treadmilling speed; further indicating the prior criteria of maximizing the time spent on the filament to be insufficient. Therefore, a threshold is reach when cargo detachment rate is larger than attachment rate such that for optimal cargo transport, filament treadmilling speed becomes dependent on the networks density.

3.4 Discussion

Our work's focus is to understand the influence of a time-dependent cytoskeletal network on cargo transportation. For static networks, cargo transport properties have been well studied [3,17]. In a real system, however, networks are not static, and we have shown how the transportation behaviors of cargo is influenced by dynamical networks, specifically for actin filaments that treadmill. There is an optimal filament treadmilling speed that enhances the mean-squared displacement (MSD) of the cargo and optimizes the mean first-passage time (MFPT). Close to the optimal speed, the time that the cargo spends on the filament is increased and the transportation is enhanced. The optimal filament speed lies within *in vivo* filament treadmilling speeds, $20 - 160\ \frac{nm}{s}$. The value of the optimal speed can also be tuned by attachment and detachment rates. Our work has implications for the efficient delivery of critical cargos to specific locations in cells. For example, cells can tune both filament treadmilling speeds and attachment/detachment rates, allowing for fine-tuned optimal transport of different cargos under different conditions. To validate our results, it would also be interesting for *in vitro* experiments to be performed to measure myosin transport

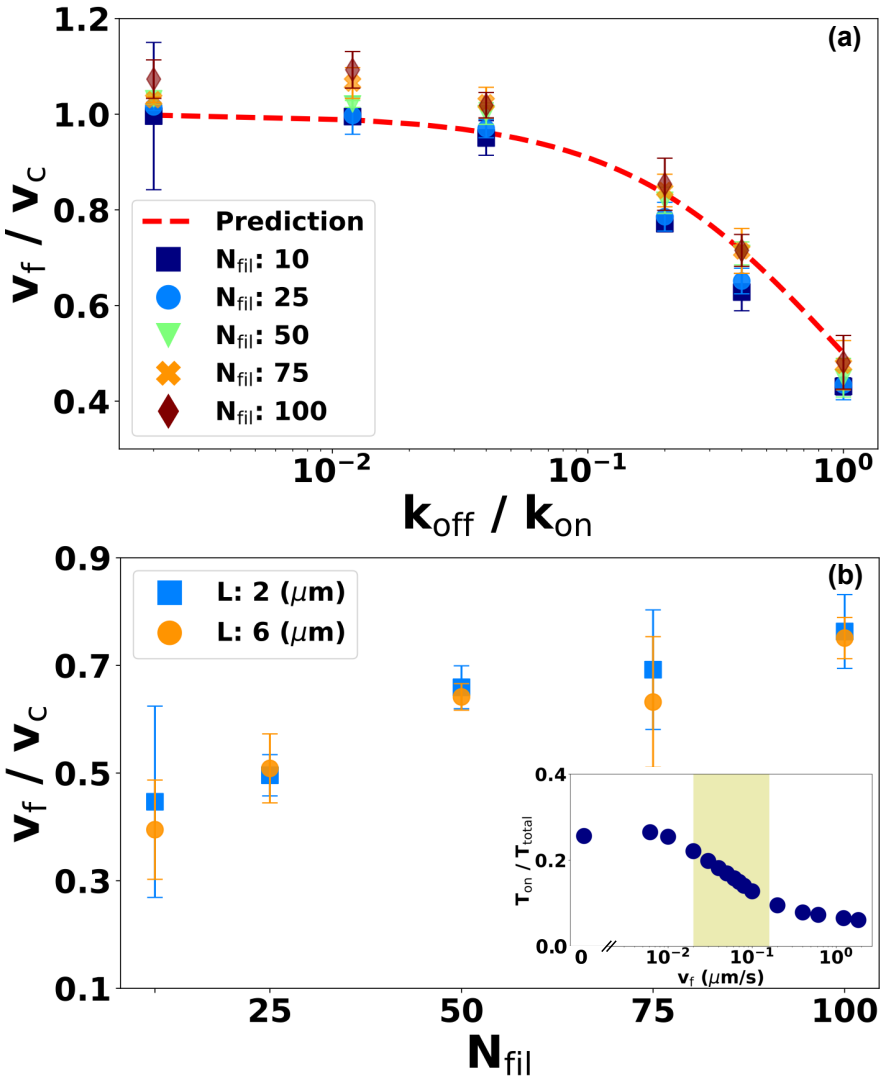

Figure 3.4: (a) Ratio of optimum filament speed to cargo speed vs ratio of detachment to attachment rates. Red-dashed line represents the analytical optimal filament speed for $k_{off} \leq k_{on}$. (b) For $k_{off} > k_{on}$, ratio of optimum filament to cargo speed vs number of filaments are shown. Inset: Fraction of time spent by cargo on filament shows monotonic behavior.

on dynamic actin filaments where the dynamics can be tuned. Real cells have well-defined network topologies, so there will be some interplay between dynamics and geometry, potentially a future study. Much work is still required to understand the depth of impact for dynamic actin networks on intracellular transport. Given our threshold case of detachment rate larger than attachment rate, it remains unknown how the filament treadmilling speed begins to have a dependence on network density. The preliminary analysis is worked out in the Appendix §3.5, which is left for future

studies. For the future work, it would be interesting to answer, how do different network arrangements affect our optimal dynamics? How do cargo switch rates affect optimal dynamics?

3.5 Appendix

3.5.1 Simulation Parameters

Shown below are the common parameter values for our system. We have used most of them in previous work [3,7,17]. The actin filaments will have a range of growth/decay speed along with the number of filaments; whereas for the cargo, it will have a range of detachment rates which we will explore.

3.5.2 Tuned Filament Speed Time On and Distance Traveled

A cargo can attach to a filament with an attachment rate of k_{on}; conversely it can detach from a filament with a detachment rate of k_{off}. When a cargo is attached to a filament, it walks for a time $\tau_b = \frac{1}{k_{off}}$. If in the diffusing phase, cargo will diffuse for a time until it is it's radius, c_{rad}, away from the filament $\tau_u = \frac{1}{k_{on}}$. The time it takes the cargo to move a distance c_{rad} away from the filament is $\tau_c = \frac{a^2}{D}$, where a is the diffusive distance in μm and D is the diffusion constant with units $\frac{\mu m^2}{s}$. In our case, since the diffusive distance is $a = 0.1\ \mu m$, the time τ_c is the time required for the cargo to diffuse "a" away from the filament. Probability of the cargo to escape from the filament, so it remains in the diffusive phase, is $P_{esc} = e^{-k_{on}\tau_c}$, where k_{on} is the attachment rate with units s^{-1}. Additionally, the types of events that the cargo can have, while on the filament, is to walk, fall-off then rebind. Therefore, the number of times these events can occur before it escapes, (cargo can walk, fall-off and rebind to the filament), is proportional to the reciprocal of the escape probability, $N_{esc} \propto \frac{1}{P_{esc}} = e^{k_{on}\tau_c}$.

From (3.1), we know that the optima filament speed can be tuned. This relationship allows the investigation of behaviors associated with time, such as how long the cargo stays on the filament. Additionally, behaviors corresponding to distance, such as the step-length or total distance traveled while on the filament. To start, we have to consider three different ranges for the filaments movement speed; first is the

Parameter Name [Units]	**Value(s)**	**Description**
t_{steps}	90,000	Number of time-steps for simulation
$t_{physical}$ $[s]$	0.049	Physical time per step [3, 17]
t_{total} $[s]$	4410	Total simulation time
D $[\frac{\mu m^2}{s}]$	0.001	Diffusion constant [7]
a $[\mu m]$	0.014	Diffusion distance
N_{cargo}	100	Cargo's simulated
N_{net}	20	Networks realized by each cargo
N_{fil}	10, 25, 50, 75, 100	Number of filaments in network
L $[\mu m]$	2, 6	Filament lengths
k_{on} $[s^{-1}]$	5.0	Cargo attachment rate [3, 17]
k_{off} $[s^{-1}]$	0.01, 0.06, 0.20, 1.0, 2.0, 5.0, 10.0	Cargo detachment rate [7]
v_c $[\frac{\mu m}{s}]$	0.0612	Cargo ballistic speed [7]
v_f $[\frac{\mu m}{s}]$	0.0, 0.00612, 0.01, 0.02, 0.03, 0.04, 0.05, 0.0612, 0.07, 0.08, 0.1, 0.2, 0.4, 0.612, 1.224, 1.836	Filament speed [7]

Table 3.1: Simulation parameter values. The diffusion distance was calculated using $a = \sqrt{4Dt_{physical}}$.

control case where our network is static ($v_f = 0$), second is the optimal case when the cargo and filament network co-move together ($v_f = v_f^o$), and third is the limit case when the filament moves much faster than the cargo ($v_f \gg v_c$). As long as the cargo is associated with the same filament of length L, the possible events are walk, fall-off and rebind. Therefore, we have the following number of events relation, for the three filament regimes, before the cargo escapes the filament with $\tau_c = \frac{a^2}{D}$,

$$N_{esc} = \begin{cases} \frac{L}{2v_c} k_{off}, & \text{for} \quad v_f = 0 \\ e^{k_{on}\tau_c}, & \text{for} \quad v_f = v_f^o \\ \frac{L}{2v_f} k_{off}, & \text{for} \quad v_f \gg v_c \end{cases} \tag{3.2}$$

Cargo walk towards either the positive end if carried by kinesin/myosin motors or negative end for dynein from the filaments midpoint, hence the division by 2 in the static ($v_f = 0$) and large speed ($v_f \gg v_c$) cases. To investigate the time and distance affects in filament speeds three regions, we start by stating the general equations (3.3). The time the cargo spent on the filament is given by T_{on} (equation (3.3a)) and the distance traveled on the filament by l (equation (3.3b)).

$$T_{on} = N_{esc}\left(\tau_b + \tau_u\right) \tag{3.3a}$$

$$l = N_{esc} v_c \tau_b = N_{esc} \frac{v_c}{k_{off}} \tag{3.3b}$$

In the static case ($v_f = 0$), the filament network does not move and all three event types can happen (walk, fall-off, rebind). The corresponding time on the filament and distance traveled is given in equation (3.4).

$$T_{on} = \frac{L}{2v_c} k_{off} \left(\frac{1}{k_{off}} + \frac{1}{k_{on}} \right) \tag{3.4a}$$

$$l = \frac{L}{2} \tag{3.4b}$$

In the optimal case ($v_f = v_f^o$), the cargo and filament co-move together (while attached). Assuming it's far from the filament ends and a low detachment rate, the event we see is just walking. Additionally, if the cargo attaches to the filament at an endpoint, we can expect events where we see it fall-off and rebind causing trailing

affects to occur. The corresponding time on the filament and distance traveled is given in equation (3.5).

$$T_{on} = e^{k_{on}\frac{a^2}{D}}\left(\frac{1}{k_{off}} + \frac{1}{k_{on}}\right) \tag{3.5a}$$

$$l = e^{k_{on}\frac{a^2}{D}}\frac{v_c}{k_{off}} \tag{3.5b}$$

In the large speed case ($v_f \gg v_c$), cargo only sees the negative end of the filament. Once it unbinds, it will always be at least c_{rad} away. Therefore, only time spent on the filament is important and the term with τ_u in equation (3.3a) is disregarded. The corresponding time on the filament and distance traveled is given in equation (3.6). Due to the large speed of the filament, it is very unlikely to see any trailing affects.

$$T_{on} = \frac{L}{2v_f} \tag{3.6a}$$

$$l = \frac{L}{2}\frac{v_c}{v_f} \tag{3.6b}$$

3.5.3 Density Dependent Optimal Filament Speed

If we consider that the density of our network is important, then the fraction of time a cargo spends on the filament network is no longer maximized. Therefore, the optimal filament speed assumptions will need to be modified. To approach this, we want to find a regime to be on the filament such that transportation is optimal. Therefore, it makes sense to work with the fraction of total time the cargo spends actually walking on the filament, defined as F_{on} = (ratio total time on) · (ratio of characteristic time on), shown in equation (3.7), with T_{diff} as total time diffusing and T_{fil} total time on the filament. If the filament moves to fast, then the cargo will encounter many filaments. Similarly, if the detachment rate is much larger than the attachment rate, $k_{off} \gg k_{on}$, then cargo will encounter more filaments. This motivates us to work with $k_{off} \gg k_{on}$, such that the cargo has a chance to encounter more filaments while the filament speed is in the optimal region, $v_f = v_f^o$, but also modifies the time spent on the filament, from equation (3.3a), which simplifies to

$T_{on} \approx \tau_b \cdot N_{esc} = \tau_b \cdot e^{k_{on}\tau_c}$.

$$F_{on} = \left(\frac{T_{fil}}{T_{fil} + T_{diff}}\right) \cdot \left(\frac{k_{on}}{k_{off}}\right) \tag{3.7}$$

Starting in the diffusive phase, we want to know if there is a dependence between the cargo finding a filament and the filament speed. If the cargo wonders into a tube, area surrounding the filament, then there exist a finite chance of running into the filament. While our cargo wonders, an tube-like area is swept out by the filament treadmilling. Time the cargo spends in this tube is $\tau_c = \frac{a^2}{D}$. Consider a rectangle, the filament moves in straight line with a distance $L + v_f\tau_c$ and has an attachment range of $2 \cdot c_{rad}$. The maximum distance a filament can move is constrained by the cells diameter, $2R$. Therefore the probability a cargo runs into a filament, within the filaments tube range in time, is the ratio of the filament distance to the cells diameter, equation (3.8).

$$\Pr_{tube} = \frac{L + v_f\tau_c}{2R} \tag{3.8}$$

Having the probability of a cargo running into a filament allows us to solve for the total time it diffuses, T_{diff}. Consider the cell with radius R and a specified number of filaments, N_{fil}. The separation between each filament tube, or the distance a cargo can diffuse, is $d_{fil} = \frac{R}{N_{fil}}$. Thus, the corresponding diffusive time is then $\tau_{diff} = \frac{d_{fil}^2}{D}$, where D is the diffusion constant. Therefore, the time a cargo can diffuse is given by equation (3.9), where $T_{diff} = \tau_{diff} \cdot \frac{1}{\Pr_{tube}}$.

$$T_{diff} = \frac{2R^3}{DN_{fil}^2} \cdot \frac{1}{L + v_f\tau_c} \tag{3.9}$$

From equation (3.9), we see that there exist a dependence on filament density and speed. T_{diff} is considered as the time a cargo diffuses between the tubes; however, it can also be interpreted as the time it takes the filaments end to reach the cargo. Equation (3.9) shows that changing the filament speed, v_f, changes the total time the cargo diffuses T_{diff}, small v_f will yield large T_{diff} and large v_f gives small T_{diff}. If we consider that the total time a cargo diffuses, T_{diff}, is equal to the time spent on the filament, $T_{on} \approx \tau_b \cdot N_{esc} = \tau_b \cdot e^{k_{on}\tau_c}$, then we are able to solve for the optimal filament speed, $v_f = \widetilde{v}_f$, given in equation (3.10). The new expression for optimal

filament speed is now dependent on the network density.

$$\widetilde{v}_f = \frac{1}{\tau_c} \cdot \left(k_{on} e^{-k_{on}\tau_c} \cdot \frac{2R^3}{DN_{fil}^2} - L \right) \tag{3.10}$$

The optimal filament speed found from the diffusive phase needs to coincide with the ballistic phase. At this point, the relationship is incomplete, but we still go over a starting process to find the time our cargo spends on the filament, T_{fil}. To complete the relation, the time our cargo spends on the filament will need to consider walking on the filament and trailing due to the filament ends, where it can attached and fall off, continuing that cycle.

Consider a cargo walking on the filament, near its middle $\left(\frac{L}{2}\right)$. In this phase, the cargo can switch to the diffusive phase but also reattach to the filament. We want to calculate the time our filaments end reaches the cargo. If our cargo detaches, it spends some time, $\tau_u = \frac{1}{k_{on}}$, diffusing until it reattaches to the filament. In this cycle, the distance between our cargo and the filaments end is then $\widetilde{d} = v_f\tau_u - \left(v_c - v_f\right)\tau_b = v_f\left(\tau_u + \tau_b\right) - v_c\tau_b$. Given that we start near the middle, we can solve for the time our cargo spends on the filament, T_{fil}, equation (3.11).

$$T_{fil} = \frac{L/2}{\widetilde{d}}\left(\tau_u + \tau_b\right) = \frac{L}{2}\frac{\left(\tau_u + \tau_b\right)}{v_f\left(\tau_u + \tau_b\right) - v_c\tau_b} \tag{3.11}$$

If we consider a mean filament speed, $v_f = \widetilde{v}_f = \frac{L/2}{T_{fil}}$, we get the following expression,

$$\widetilde{v}_f = \frac{\widetilde{d}}{\left(\tau_u + \tau_b\right)} \tag{3.12}$$

Thus far, filament trailing effects need to be incorporated into the theory, which is left for the future direction in this study.

Chapter 4

Final Discussion

4.1 Overall Conclusion and Future Work

The objective of this dissertation was to understand how geometry and dynamics affect anomalous diffusion. The work discussed two specific projects. The first project studied geometry, with Lévy walks in curved space, where we simulated a random walker on manifolds, to understand how the intrinsic curvature of the manifold influenced transport by studying the mean-squared displacement (MSD). This work was presented in chapter two, which showed that negatively curvature manifolds, i.e., hyperboloid, have a positive impact on transportation such that the random walker covers more spatial distance; whereas manifolds with positive curvature, i.e., a sphere, have negative impact on transport where the walker covers less of the surface when compared to transport on flat (Euclidean) manifolds. The second project focused on intracellular transport on dynamic actin networks. We considered a random walker/-cargo with two phases of motion, either diffusive in the cell's cytoplasm or ballistic on the network of actin filaments which can be thought of as tracks. Dynamics were introduced by allowing the filament to treadmill. This allowed us to analyze how the cargo spread through the cell from the nucleus by studying the MSD. In addition, optimal transport of cargo within the cell was also analyzed by studying the mean first-passage times (MFPTs). Our work, presented in chapter three, showed that there exist optimal dynamics in the network of filaments that can enhance transport.

In our work on Lévy walks in curved spaces, we developed a method to simulate random walks in curved spaces. We derived a mean-squared displacement relation

for Lévy walks in curved spaces with arbitrary Gaussian curvature that for Brownian motion matches with existing results [41]. We showed that our general expansion works when not in the Brownian regime. The MSD decreases for spherical surfaces but increases for hyperbolic. The unknown coefficients in the expansion were also estimated. Interestingly, the coefficients are on the order of 1, and is insensitive to the Lévy exponent, μ, but this requires more investigation. We showed that curved surfaces will produce an apparent shift in μ unless you account for curvature properly. For example, on a sphere, Lévy type motion, $2 < \mu < 3$, could appear Brownian, with $\mu > 3$, if you don't account for the curvature corrections. Overall, we were able to account for geometrical effects on anomalous diffusion.

Our results can be validated by measuring the diffusion of membrane bound proteins on lipid membranes that are supported on engineered surfaces with different curvatures. Our work has implications for processes occurring in cells. For example, membrane signaling and reaction rates depend on diffusion, which we have shown depends on local surface curvature. So, changes in curvature can lead to changes in these rates which in turn lead to changes in the corresponding biological process. Since our work is novel, more analysis is needed to understand how geometry affects transport. For future work, we can ask, can we derive a closed form expression for the expansion coefficients? How are optimal search processes affected by curvature? Can we build a pipeline to analyze experimental data of membrane protein diffusion to extract the anomalous exponent, diffusion constant and local curvature?

In chapter 3 we have shown how the transportation behaviors of cargo is influenced by dynamical networks, specifically for actin filaments. There is an optimal filament treadmilling speed that enhances the mean-squared displacement of the cargo and optimizes the mean first-passage time. Close to the optimal speed, the time that the cargo spends on the filament is increased and the transportation is enhanced. The optimal filament speed lies within *in vivo* filament treadmilling speeds. The value of the optimal filament speed can also be tuned by changing the cargos attachment and detachment rates. Our work has implications for the efficient delivery of critical cargos to specific locations in cells. For example, cells can tune both filament treadmilling speeds and attachment/detachment rates, allowing for fine-tuned optimization of transport of different cargo under different conditions.

To validate our results, it would also be interesting for *in vitro* experiments to

be performed to measure myosin transport on dynamic actin filaments where the dynamics can be tuned. Real cells have well-defined network topologies, so there will be some interplay between dynamics and geometry, potentially a future study. For the future work, we can ask, how do different network arrangements affect our optimal dynamics? As an example for radial networks that favor growth/shrinkage on one end, such as for microtubules, and polarized such that the positive end of the filament points towards the cells outer membrane, we have derived a MFPT function that relates the density of the network to transportation time, shown in section §A.1 of the Appendix. In addition, statistical variations need to be tested for the work shown in chapter three. Work from [17] showed that cargo-to-cargo variation, e.g., attachment/detachment rate, influences transportation more compared to network-to-network variations, e.g., filament length or density.

Overall, our results have shed light on the geometry and dynamics on the substrates taken place in their respective environments. We hope this will lead to more detailed studies and experimental validations for our results.

Appendix A

Appendix: Future work Derivations

A.1 Reaction-Rate Derivation: Linear Potential

From Kramers' theory, we are able to describe the crossing from diffusive barriers. This allows us to model a diffusing particle with drift in a potential [52]. Moreover, the mean first-passage time, (MFPT), can be computed with a potential. Following the work from [52], with the diffusion constant, $D = \frac{k_B T}{M\gamma} = (\gamma M \beta)^{-1}$, where k_B is the Boltzmann constant, T the (thermodynamic) temperature, γ a constant damping rate, M as the particle mass, and a potential $U(x)$, the MFPT, $t(x)$, for a one-dimensional Smoluchowski equations is,

$$-1 = -(M\gamma)^{-1} U^{'}(x) t^{'}(x) + (M\gamma)^{-1} k_B T t^{''}(x) \tag{A.1}$$

The relationship between $U^{'}(x)$ and drift velocity v can be connected through the fluctuation-dissipation theorem (FDT). A particle walks through a region of fluid such that it experiences some drag. Dissipating kinetic energy, in that region, into heat or fluctuations in diffusive motion. This causes a drift, v, which is related to a constant mobility (μ) and force ($F(x) = U^{'}(x)$), thus $v = \mu F(x) = \mu U^{'}(x)$. In addition, the mobility is also related to drag, $\mu = (M\gamma)^{-1}$. Using the Einstein-Smoluchowski relation, we get the diffusion coefficient, $D = \mu k_B T = \mu/\beta$. Therefore, by replacing mobility, $\mu = v/U^{'}(x)$, the diffusion coefficient then becomes

$D = \mu k_B T = v/\left(\beta U^{'}(x)\right)$. This states that the force applied to our particle in this region is constant, $\beta U^{'}(x) = v/D$, therefore $\beta U^{'}(x)$ is linearly proportional to distance x.

Consider a system with a dense network, polarized such that the growing/shrinking end points towards the cell membrane while the negative end remains static. This is shown in Figure A.1 [3]. We start with the potential energy defined by equation (A.2), such that we have a network that does not remain fixed in length and has a growth speed of v. The potential equations was motivated by [3], which represents a radial drift in the region, that is spherically symmetric, of filaments such that there exist a linear potential, $U(r) \propto r$, and a constant potential outside the region of filaments, (C_1, C_2), where no drift exists. The transition from these different regions should be considered as diffusive barrier crossings.

$$\beta U(r) = \begin{cases} C_1, & r_n \leq r \leq r_a, \\ C_1 - \frac{v}{D} \cdot (r - r_a), & r_a \leq r \leq r_a + w, \\ C_2 = C_1 - \frac{v}{D} \cdot w, & r_a + w \leq r \leq R \end{cases} \tag{A.2}$$

This defines a system starting from the nucleus, radius $R_n = r_n$, to the first filament start position, $R_a = r_a$, where the potential is constant, C_1. The next region has filaments, with length w, and starts from R_a to $R_a + w$, which has drift, modeled by a linear potential that is dependent on the radial position from the nucleus. The last region starts from the end of the filament, $R_a + w$, to the cell membrane, R, where no drift occurs and has a constant potential C_2. This system holds particularly for microtubules with radial arrangements such that the positive end points towards the cell membrane, whereas the negative end point towards the nucleus. In addition, since the negative end is insignificant in dynamics compared to the positive end, we can assume the positive end of the filament to be dynamic, therefore the negative end remains static. From [52], we have the following mean first-passage time (MFPT) equation as a function of radial distance r, $t(r)$, with r_n reflecting and R absorbing boundaries.

$$t(r) = \frac{1}{D} \int_r^R d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) \tag{A.3}$$

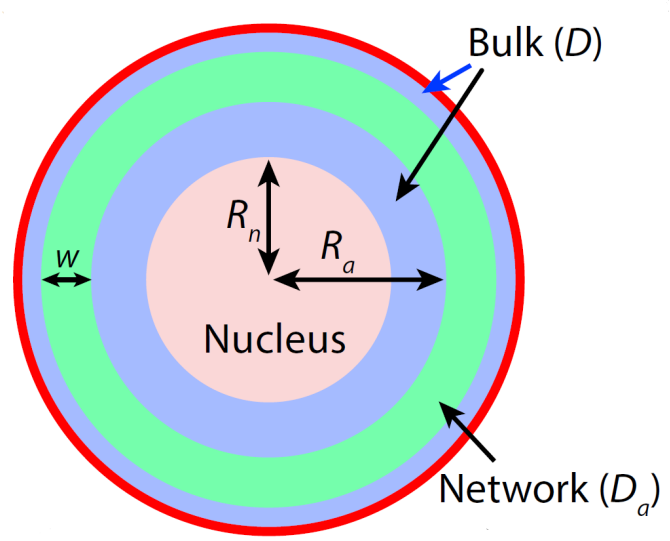

Figure A.1: Depicted is a eukaryotic cell as a sphere. Here, the nucleus has a radius of R_n, with the filaments in green, having length w such that the starting end is at a radius R_a away from the nucleus center. The blue represents the cytoplasm with bulk diffusion D. In this depiction, the network of filaments in green can be considered part of the cytoplasm with a network diffusion constant D_a [3].

Applying our potential energy equation (A.2), the MFPT integral will have the following form:

$$\begin{aligned} t(r) = &\frac{1}{D}\left\{\int_{r_n}^{r_a} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right\} \\ &+ \frac{1}{D}\left\{\int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right\} \\ &+ \frac{1}{D}\left\{\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right\} \end{aligned} \tag{A.4}$$

For simplicity, we will express equation (A.4) as:

$$t(r) = t_1 + t_{1,2} + t_{1,2,3} \tag{A.5}$$

The solutions to equation (A.4) for the respective time will be done in separate pieces in the following subsections, For t_1 in section §A.1.1, in the middle region for $t_{1,2}$ in section §A.1.2 and in the last region for $t_{1,2,3}$ in section §A.1.3.

A.1.1 Solving t_1

We start this section by solving the first region where the time is defined by t_1 with the following form,

$$\begin{aligned} t_1 &= \frac{1}{D}\left\{\int_{r_n}^{r_a} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right\} \\ &= \frac{1}{D}\left\{\int_{r_n}^{r_a} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \left[\int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right]\right\} \end{aligned}$$

Applying the corresponding potential energy,

$$\begin{aligned} t_1 &= \frac{1}{D}\int_{r_n}^{r_a} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \left[\int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right] \\ &= \frac{1}{D}\int_{r_n}^{r_a} d\rho \cdot \frac{\exp(C_1)}{\rho} \cdot \left[\int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-C_1)\right] \\ &= \frac{1}{D}\int_{r_n}^{r_a} d\rho \cdot \frac{1}{\rho} \cdot \left[\int_{r_n}^{\rho} d\sigma \cdot \sigma\right] \\ &= \frac{1}{D}\int_{r_n}^{r_a} d\rho \cdot \frac{1}{\rho} \cdot \left[\frac{1}{2} \cdot \left(\rho^2 - r_n^2\right)\right] \\ &= \frac{1}{2 \cdot D}\int_{r_n}^{r_a} d\rho \cdot \left(\rho - \frac{r_n^2}{\rho}\right) \\ &= \frac{1}{2 \cdot D}\left[\frac{1}{2} \cdot \rho^2 - r_n^2 \cdot \ln(\rho)\right]\Big|_{r_n}^{r_a} \\ &= \frac{1}{2 \cdot D}\left[\frac{1}{2} \cdot \left(r_a^2 - r_n^2\right) - r_n^2 \cdot \ln\left(\frac{r_a}{r_n}\right)\right] \end{aligned}$$

Therefore, the first right-hand-side (RHS) term of equation (A.5) evaluates to,

$$t_1 = \frac{1}{2 \cdot D}\left[\frac{1}{2} \cdot \left(r_a^2 - r_n^2\right) - r_n^2 \cdot \ln\left(\frac{r_a}{r_n}\right)\right] \tag{A.6}$$

A.1.2 Solving $t_{1,2}$

This section solves the middle region where the time is defined by $t_{1,2}$ with the following form,

$$\begin{aligned} t_{1,2} &= \frac{1}{D}\left\{ \int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) \right\} \\ &= \frac{1}{D}\Bigg\{ \int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \\ &\qquad \left[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \int_{r_a}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) \right] \Bigg\} \end{aligned}$$

Applying the corresponding potential energy and canceling out constants. Note

the expression of $\exp(C_1)$ will always cancel.

$$
\begin{aligned}
t_{1,2} &= \frac{1}{D}\Bigg\{ \int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \\
&\qquad \left[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \int_{r_a}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) \right] \Bigg\} \\
&= \frac{1}{D}\Bigg\{ \int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(C_1 - \frac{v}{D}(\rho - r_a))}{\rho} \cdot \\
&\qquad \left[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-C_1) + \int_{r_a}^{\rho} d\sigma \cdot \sigma \cdot \exp(-C_1 + \frac{v}{D}(\sigma - r_a)) \right] \Bigg\} \\
&= \frac{1}{D}\Bigg\{ \int_{r_a}^{r_a+w} d\rho \cdot \frac{\exp(-\frac{v}{D}(\rho - r_a))}{\rho} \cdot \\
&\qquad \left[\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v}\left((\rho - \frac{D}{v}) \cdot \exp\left(\frac{v}{D}(\rho - r_a) \right) + (\frac{D}{v} - r_a) \right) \right] \Bigg\} \\
&= \frac{1}{D}\Bigg\{ \int_{r_a}^{r_a+w} d\rho \cdot \Bigg[\left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v} \cdot (\frac{D}{v} - r_a) \right) \cdot \frac{\exp\left(-\frac{v}{D}(\rho - r_a)\right)}{\rho} + \\
&\qquad \frac{D}{v} \cdot \left(1 - \frac{D}{v} \cdot \frac{1}{\rho} \right) \Bigg] \Bigg\} \\
&= \frac{1}{v}\left[\rho - \frac{D}{v} \cdot \ln(\rho) \right] \Big|_{r_a}^{r_a+w} + \Bigg\{ \frac{1}{D} \cdot \left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v} \cdot (\frac{D}{v} - r_a) \right) \cdot \\
&\qquad \int_{r_a}^{r_a+w} d\rho \cdot \left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a)\right)}{\rho} \right] \Bigg\} \\
&= \frac{1}{v}\left[w - \frac{D}{v} \cdot \ln\left(\frac{r_a + w}{r_a} \right) \right] + \Bigg\{ \frac{1}{D} \cdot \left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v} \cdot (\frac{D}{v} - r_a) \right) \cdot \\
&\qquad \int_{r_a}^{r_a+w} d\rho \cdot \left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a)\right)}{\rho} \right] \Bigg\}
\end{aligned}
$$

Thus, the second RHS term of equation (A.5) evaluates to,

$$
\begin{aligned}
t_{1,2} = \frac{1}{v}\left[w - \frac{D}{v} \cdot \ln\left(\frac{r_a + w}{r_a} \right) \right] &+ \Bigg\{ \frac{1}{D} \cdot \left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v} \cdot (\frac{D}{v} - r_a) \right) \cdot \\
&\int_{r_a}^{r_a+w} d\rho \cdot \left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a)\right)}{\rho} \right] \Bigg\}
\end{aligned} \tag{A.7}
$$

The last term is called the exponential integral, usually abbreviated as $Ei(r)$, and will need numerical integration to solve it.

A.1.3 Solving $t_{1,2,3}$

This section solves the last region where the time is defined by $t_{1,2,3}$ with the following form,

$$\begin{aligned} t_{1,2,3} &= \frac{1}{D}\left\{\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \int_{r_n}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right\} \\ &= \frac{1}{D}\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \left[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \right. \\ &\qquad \left. \int_{r_a}^{r_a+w} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \int_{r_a+w}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma))\right] \end{aligned}$$

Applying the corresponding potential energy and canceling out constants. Note the expression of $\exp(C_1)$ will always cancel. After the first integral, we combine the constants together into one constant.

$$
\begin{aligned}
t_{1,2,3} &= \frac{1}{D}\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(\beta U(\rho))}{\rho} \cdot \Bigg[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \\
&\qquad \int_{r_a}^{r_a+w} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) + \int_{r_a+w}^{\rho} d\sigma \cdot \sigma \cdot \exp(-\beta U(\sigma)) \Bigg] \\
&= \frac{1}{D}\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(C_1 - \frac{v}{D}\cdot w)}{\rho} \cdot \Bigg[\int_{r_n}^{r_a} d\sigma \cdot \sigma \cdot \exp(-C_1) + \\
&\qquad \int_{r_a}^{r_a+w} d\sigma \cdot \sigma \cdot \exp(-C_1 + \frac{v}{D}(\sigma - r_a)) + \\
&\qquad \int_{r_a+w}^{\rho} d\sigma \cdot \sigma \cdot \exp(-C_1 + \frac{v}{D}\cdot w) \Bigg] \\
&= \frac{1}{D}\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(-\frac{v}{D}\cdot w)}{\rho} \cdot \Bigg[\int_{r_n}^{r_a} d\sigma \cdot \sigma + \\
&\qquad \int_{r_a}^{r_a+w} d\sigma \cdot \sigma \cdot \exp(\frac{v}{D}(\sigma - r_a)) + \int_{r_a+w}^{\rho} d\sigma \cdot \sigma \cdot \exp(\frac{v}{D}\cdot w) \Bigg] \\
&= \frac{1}{D}\int_{r_a+w}^{R} d\rho \cdot \frac{\exp(-\frac{v}{D}\cdot w)}{\rho} \cdot \Bigg[\frac{1}{2}(r_a^2 - r_n^2) + \\
&\qquad \frac{D}{v}\Bigg[\exp(\frac{v}{D}\cdot w) \cdot \left(r_a + w - \frac{D}{v} \right) + \left(\frac{D}{v} - r_a \right) \Bigg] + \\
&\qquad \exp(\frac{v}{D}\cdot w) \cdot \left(\frac{1}{2}(\rho^2 - (r_a+w)^2) \right) \Bigg] \\
&= \frac{1}{D}\int_{r_a+w}^{R} d\rho \left[\left(X_T \cdot \frac{1}{\rho} \right) + \left(\frac{1}{2}\cdot \rho \right) \right] \\
&= \frac{1}{D}\left[X_T \cdot \ln(\rho) + \frac{1}{4}\rho^2 \right] \Big|_{r_a+w}^{R} \\
&= \frac{1}{D}\left[X_T \cdot \ln\left(\frac{R}{r_a+w} \right) + \frac{1}{4}\left(R^2 - (r_a+w)^2 \right) \right]
\end{aligned}
$$

where $X_T = \Bigg[\frac{1}{2}\exp(-\frac{v}{D}\cdot w)\cdot(r_a^2 - r_n^2) + \frac{D}{v}\Bigg[\left(r_a + w - \frac{D}{v}\right) + \exp(-\frac{v}{D}\cdot w)\cdot\left(\frac{D}{v} - r_a\right) \Bigg] - \left(\frac{1}{2}(r_a+w)^2\right) \Bigg]$ is a constant.

Thus, the third RHS term of equation (A.5) evaluates to,

$$t_{1,2,3} = \frac{1}{D}\left[X_T \cdot \ln\left(\frac{R}{r_a+w}\right) + \frac{1}{4}\left(R^2 - (r_a+w)^2\right)\right] \tag{A.8}$$

A.1.4 Final MFPT Form: Linear Potential

Putting everything together, the solution for equation (A.3) with our potential energy (A.2) is,

$$\begin{aligned} t = {} & \frac{1}{2\cdot D}\left[\frac{1}{2}\cdot\left(r_a^2 - r_n^2\right) - r_n^2\cdot\ln\left(\frac{r_a}{r_n}\right)\right] + \frac{1}{v}\left[w - \frac{D}{v}\cdot\ln\left(\frac{r_a+w}{r_a}\right)\right] \\ & + \left\{\frac{1}{D}\cdot\left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v}\cdot(\frac{D}{v} - r_a)\right)\cdot\int_{r_a}^{r_a+w} d\rho\cdot\left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a)\right)}{\rho}\right]\right\} \\ & + \frac{1}{D}\left[X_T\cdot\ln\left(\frac{R}{r_a+w}\right) + \frac{1}{4}\left(R^2 - (r_a+w)^2\right)\right] \end{aligned} \tag{A.9}$$

where $X_T = \left[\frac{1}{2}\exp(-\frac{v}{D}\cdot w)\cdot(r_a^2 - r_n^2) + \frac{D}{v}\left[\left(r_a + w - \frac{D}{v}\right) + \exp(-\frac{v}{D}\cdot w)\cdot\left(\frac{D}{v} - r_a\right)\right] - \left(\frac{1}{2}(r_a+w)^2\right)\right]$ is a constant.

MFPT Diffusion only

If we consider no filaments in our system, we get pure diffusion. The only term that matters is then equation (A.6) but from r_n to R. Therefore, the MFPT in pure diffusion is,

$$t = \frac{1}{2\cdot D}\left[\frac{1}{2}\cdot\left(R^2 - r_n^2\right) - r_n^2\cdot\ln\left(\frac{R}{r_n}\right)\right] \tag{A.10}$$

A.1.5 MFPT, Linear Potential: Velocity Rate of Change

For numerical purposes, we write the first derivative of equation (A.9) with respect to velocity.

$$\begin{aligned}
\frac{dt}{dv} = & \frac{1}{v^2} \cdot \left[2 \cdot \frac{D}{v} \cdot \ln\left(\frac{r_a + w}{r_a} \right) - w \right] \\
& - \left\{ \frac{w}{D^2} \cdot \left(\frac{1}{2}(r_a^2 - r_n^2) + \frac{D}{v} \cdot (\frac{D}{v} - r_a) \right) \cdot \int_{r_a}^{r_a + w} d\rho \cdot \left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a) \right)}{\rho} \right] \right\} \\
& + \frac{1}{v^2} \cdot \left(r_a - 2 \cdot \frac{D}{v} \right) \cdot \int_{r_a}^{r_a + w} d\rho \cdot \left[\frac{\exp\left(-\frac{v}{D}(\rho - r_a) \right)}{\rho} \right] \\
& + \frac{1}{D} \cdot \ln\left(\frac{R}{r_a + w} \right) \cdot \left[\frac{-w}{2 \cdot D} \cdot \left(r_a^2 - r_n^2 \right) \cdot \exp\left(-\frac{v}{D} \cdot w \right) \right] \\
& + \ln\left(\frac{R}{r_a + w} \right) \cdot \left[\frac{1}{v^2} \cdot \left(2 \cdot \frac{D}{v} - (r_a + w) \right) \right] \\
& + \ln\left(\frac{R}{r_a + w} \right) \cdot \left[\frac{1}{v} \cdot \exp\left(-\frac{v}{D} \cdot w \right) \cdot \left(\frac{1}{v} \cdot \left(r_a - 2 \cdot \frac{D}{v} \right) + \frac{r_a \cdot w}{D} - \frac{w}{v} \right) \right]
\end{aligned} \tag{A.11}$$

Appendix B

Appendix: Computer Programs Used

B.1 Introduction

In this work, the first projects simulation, numerical calculations and analysis were executed in MATLAB; whereas in the second project the simulations were executed using C and all resulting data was analyzed using Python. Here, we show the programs used to conduct the different research projects. The full code of simulation and analysis can be found on github, click (**Imtiaz_Ali_Dissertation_Code**). If the link does not work, the website url is:

- https://github.com/imtiaza20/Imtiaz_Ali_Dissertation_Code.git

B.2 Lévy Walks in Curved Space Programs

B.2.1 LWALK_IA_SPHERE_MAIN_DATA.m

This program simulates the Lévy random walk motion in Spherical Space.

```
% Imtiaz Ali, Physics Graduate Student, Univeristy Of
   California, Merced
% Description: Levy walk on a surface of sphere using the
   method of quaternion. Walks are
```

```
%              discrete step-lengths.

% ############################################################
% ############################################################

% To perform a random walk on the sphere, we need to apply the
    following steps:
% This is done for each step/levy walk
% 1. -> Need to compute the normalized vector to rotate about.
% 2. -> From our initial position, we pick so Theta and Phi
   angle to compute (x2,y2,z2)
% 3. -> Apply cross product to get vector to rotate about.
% 4. -> Normalize this new vector to get our random axis of
   rotation.
%    -> Note: We cannot pick a normalized vector at random
%    -> (e.g, vec_n = [rand,rand,rand]), this will cause
   issues.
% 5. -> Pick central angle sig = S/R
% 6. -> Apply Quaternion methods, (x',y',z') = q(x,y,z)q^-1 =>
    q*q^-1 = 1
%    -> Note: q = (cos(sigma/2),sin(sigma/2)*V),
%    -> V = RANDOM UNIT VECTOR sigma = S/R -> S = 1 (geodesic
   length)
%    -> Note: q^-1 = (cos(sigma/2),-sin(sigma/2)*V), (inverse
   of the quaternion)
% 7. -> This is our new updated position on the sphere.
%    -> NOTE:
%     -> If initial zenith angle is pi/2. We are performing a
   walk on the meridian.
%     -> If initial zenith angle is 0 and new random z-angle
   is 0.
%     -> We are performing a walk on the equator.
%     -> Max distance allowed is S < R*pi, => 0 < central
   angle < pi

% ############################################################
```

```
% Quaternion equations:
% -> qL = -sin(sig/2)*(vr_hat(1)*x + vr_hat(2)*y + vr_hat(3)*z
   );
% -> qM = cos(sig/2)*x + sin(sig/2)*(vr_hat(2)*z - vr_hat(3)*y
   );
% -> qN = cos(sig/2)*y + sin(sig/2)*(vr_hat(3)*x - vr_hat(1)*z
   );
% -> qP = cos(sig/2)*z + sin(sig/2)*(vr_hat(1)*y - vr_hat(2)*x
   );
%
% % Update position on sphere -> Apply Quaternion method
% -> x' = cos(sig/2)*qM + sin(sig/2)*(vr_hat(2)*qP - vr_hat(1)
   *qL - vr_hat(3)*qN);
% -> y' = cos(sig/2)*qN + sin(sig/2)*(vr_hat(3)*qM - vr_hat(1)
   *qP - vr_hat(2)*qL);
% -> z' = cos(sig/2)*qP + sin(sig/2)*(vr_hat(1)*qN - vr_hat(2)
   *qM - vr_hat(3)*qL);

% ############################################################
% ############################################################

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% tic
close all
clear

% SIMULATION PARAMETERS
samples = 1; % TOTAL NUMBER OF RUNS
rng('shuffle'); % CHANGE WHAT RAND SAMPLES
epsi_1 = 1e-6; % MACHINE PRECISION
epsi_2 = eps(0); % OR 0, LIMIT FOR SMALLEST PROBABILITY SO WE
   GET S_MAX
```

```
% DYNAMICS PARAMETERS
R = 100; % SPHERE RADIUS
L = 1; % INCREMENTAL STEP-LENGTH -> LEVY STEP-LENGTH -> DO NOT
    REMOVE
vel_walk = L; % LEVY WALK VELOCITY
mu = 3.4; % LEVY \ MU
tmax = 100; % MAX NUMBER OF DISCRETE LEVY WALKS
S_MIN = 1; % MINIMUM GEODESIC DISTANCE
b = 1; % HIGHEST PROBABILITY VALUE
a = epsi_2; % SET SMALLEST NUMBER TO BARLEY HITTING MACHINE
   PRECISION 1e-3 = 0.001
S_MAX = floor(R*2*pi); % MAX GEODESIC VALUE ROUNDED UP -> S TO
    S_MAX
FOR_EX = 0; % EXIT WHILE LOOP PARAMETER FOR LEVY WALK

% MAKE RAND RESTRICTION SO WE DONT SAMPLE CLOSE TO ZERO BUT IT
    CANT BE TO SMALL EITHER
% GENERATE RNG IN SPECIFIC INTERVAL
% -> [a,b] = [0.01,1] -> rand_vec = (b-a).*rand(1000,1) + a ->
    GET 1000 VALUES
rand_vec = (b - a).*rand(100000,1) + a;
SZE = size(rand_vec,1);

% ###############################################################
% ###############################################################

% TOTAL RUN SETUP
% -> cell(mu(n),samples) = zeros(1,1)
[mfpt_totalrun_SPH{1:size(mu,1),1:samples}] = deal(zeros(1,1))
   ;

% SEARCHER INITIALIZATION AND DATA -> ALWAYS HAS tmax + 1 ROWS
   :
% -> cell(mu(n),samples) = zeros(time-step,1)
[x_SPH_d_totalrun{1:size(mu,1),1:samples}] = deal(zeros(1,1));
```

```
[y_SPH_d_totalrun{1:size(mu,1),1:samples}] = deal(zeros(1,1));
[z_SPH_d_totalrun{1:size(mu,1),1:samples}] = deal(zeros(1,1));
[d_S_B3{1:size(mu,1),1:samples}] = deal(zeros(1,1));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% disp('START MAIN DATA LOOP');
% tic
for m = 1:samples % START ITERATION OVER ALL SAMPLES/RUNS

    % USE THIS PARAMETER AS OUR NUMBER OF TIME STEPS/INCREMETS
    mfpt = 0; % TIME-STEP INIALIZE TO ZERO, ACCEPTED UNTIL MAX
        TIME IS REACHED
    FOR_EX = 0; % EXIT WHILE LOOP PARAMETER FOR LEVY WALK

% ############################################################

    % INITIALIZE POSITION OF SEARCHER RANDOMLY
    % SPHERICAL
    phi_d_1 = rand*(2*pi); % Azimuthal
    theta_d_1 = rand*(2*pi); % Zenith
    x_SPH_d_1 = R*cos(theta_d_1)*cos(phi_d_1);
    y_SPH_d_1 = R*cos(theta_d_1)*sin(phi_d_1);
    z_SPH_d_1 = R*sin(theta_d_1);

    % INITIAL POSITION, NOT USED TO MODIFY, USED TO SET
       INITIAL OVERALL
    xtmp_SPH_d_1 = x_SPH_d_1;
    ytmp_SPH_d_1 = y_SPH_d_1;
    ztmp_SPH_d_1 = z_SPH_d_1;

    % INTIAL POSITION, WILL USE TO UPDATE STEP AS SEARCH GOES
    xtmp1_SPH_d = x_SPH_d_1;
    ytmp1_SPH_d = y_SPH_d_1;
    ztmp1_SPH_d = z_SPH_d_1;
```

```
% ############################################################

    % RECORD INITIAL STEP LOCATION ONLY -> ROW = TIME, COLUMN
       = COORDINATE VALUE
    % ALWAYS HAS tmax + 1 ROWS:
    x_SPH_d_totalrun{1,m}(mfpt+1,1) = xtmp_SPH_d_1;
    y_SPH_d_totalrun{1,m}(mfpt+1,1) = ytmp_SPH_d_1;
    z_SPH_d_totalrun{1,m}(mfpt+1,1) = ztmp_SPH_d_1;

% ############################################################
% ############################################################

    while ( mfpt < tmax ) % START ITERATION OVER ALL TIME-
       STEPS

        % LEVY WALK -> BREAK f1 INTO INCREMENTS
        % MAKE TOTAL LEVY WALK
        rand_used = rand_vec(randi([2,SZE],1),1); % PROB OF
           RANDOM LEVY STEP ds_half
        f1 = floor(((S_MAX)^(1-mu) + ((rand_used-a)/(b-a))*...
          ((S_MIN)^(1-mu)-(S_MAX)^(1-mu)))^(1/(1-mu))); % LEVY
              FLIGHT LENGTH -> CDF S TO S_MAX
        ds_SPH_temp2(1,1) = f1; % GREATER THAN S_1
        ds_SPH_temp2(2,1) = -f1; % LESS THAN S_1
        Qr_ds = randi([1,2]);
        ds_SPH = ds_SPH_temp2(Qr_ds,1); % LEVY GEODESIC LENGTH
        if ds_SPH >= (tmax - mfpt)
            ds_SPH_temp2(1,1) = (tmax - mfpt); % GREATER THAN
               S_1
            ds_SPH_temp2(2,1) = -(tmax - mfpt); % LESS THAN
               S_1
            Qr_ds = 1; % randi([1,2]);
            ds_SPH = ds_SPH_temp2(Qr_ds,1);
        end
```

```
% EXCUDE INITIAL ds_SPH_temp_MAIN_VEC POINT
% -> NO DOUBLE COUNTING OF INITIAL POSITION -> 1 TO
   ds_SPH_temp1
T_REQ = floor(abs(ds_SPH)/vel_walk); % REQUIRED TIME
   TO DO FULL LEVY FLIGHT
S_MIN_2 = L; % KEEP PROPER ITERATION FROM SIGN OF
   ds_SPH
if ds_SPH < 0
   S_MIN_2 = -L;
end

% MAKE CONDITIONAL SO WE DON'T EXCEED TMAX STEPS
% SIZE OF VECTOR NEEDS TO BE tmax - mfpt
if T_REQ >= (tmax - mfpt)
    S_MIN_2 = L;
    T_REQ = tmax - mfpt; % REQUIRED TIME TO DO FULL
       LEVY FLIGHT
    ds_SPH = T_REQ*vel_walk; % NEW DISCRETE GEODESIC
       LENGTH
end

% COMPUTE NORMALIZED VECTOR TO ROTATE ABOUT -> THIS
   METHOD IS ISOTROPIC!!
phi2 = rand*(2*pi); % AZIMUTHA, 0 < PHI < 2*PI
theta2 = rand*(pi); % ZENITH ANGLE, -PI/2 < THTA < PI
   /2
x2 = R*cos(theta2)*cos(phi2);
y2 = R*cos(theta2)*sin(phi2);
z2 = R*sin(theta2);

% MAKE UNIT VECTOR FOR AXIS OF ROTATION
vr = [(y_SPH_d_totalrun{1,m}(mfpt+1,1)*z2 -
   z_SPH_d_totalrun{1,m}(mfpt+1,1)*y2),...
  (z_SPH_d_totalrun{1,m}(mfpt+1,1)*x2 -
     x_SPH_d_totalrun{1,m}(mfpt+1,1)*z2),...
```

```
  (x_SPH_d_totalrun{1,m}(mfpt+1,1)*y2 -
     y_SPH_d_totalrun{1,m}(mfpt+1,1)*x2)];
vr_hat = (1/sqrt(vr(1)^2 + vr(2)^2 + vr(3)^2))*vr;

% MAKE DISCRETE LEVY WALKS
deltaT = T_REQ; % NUMBER OF DISCRETE LEVY WALKS

% APPLY LEVY WALK
for i = 1:deltaT % START INCREMENT OVER ALL DISCRETE
   LEVY WALK

  % CENTRAL ANGLE: NEGATIVE IS A CLOCKWISE TURN,
     POSITIVE IS A COUNTER-CLOCKWISE TURN
  sig = (S_MIN_2/R); % MAX DISTANCE S < R*PI, -> 0 <
     CENTRAL ANGLE < PI

  % APPLY QUATERNION METHOD
  % COMPUTE q*u = Q = [qL,qM,qN,qP] (COMPLEX
     MULTIPULCATION)
  qL = -sin(sig/2)*(vr_hat(1)*x_SPH_d_totalrun{1,m}(
     mfpt+1,1)...
    + vr_hat(2)*y_SPH_d_totalrun{1,m}(mfpt+1,1) +
       vr_hat(3)*z_SPH_d_totalrun{1,m}(mfpt+1,1));
  qM = cos(sig/2)*x_SPH_d_totalrun{1,m}(mfpt+1,1) +
     sin(sig/2)...
    *(vr_hat(2)*z_SPH_d_totalrun{1,m}(mfpt+1,1) -
       vr_hat(3)*y_SPH_d_totalrun{1,m}(mfpt+1,1));
  qN = cos(sig/2)*y_SPH_d_totalrun{1,m}(mfpt+1,1) +
     sin(sig/2)...
    *(vr_hat(3)*x_SPH_d_totalrun{1,m}(mfpt+1,1) -
       vr_hat(1)*z_SPH_d_totalrun{1,m}(mfpt+1,1));
  qP = cos(sig/2)*z_SPH_d_totalrun{1,m}(mfpt+1,1) +
     sin(sig/2)...
    *(vr_hat(1)*y_SPH_d_totalrun{1,m}(mfpt+1,1) -
       vr_hat(2)*x_SPH_d_totalrun{1,m}(mfpt+1,1));
```

```
            % NEXT DISCRETE LEVY WALK POSITION -> UPDATE
               POSITION ON SPHER
            % COMPUTE q*u*q^-1 = Q*q^-1 = [garbage,x',y',z'] (
               COMPLEX MULTIPULCATION)
            x_SPH_d_1 = cos(sig/2)*qM + sin(sig/2)*(vr_hat(2)*qP
                - vr_hat(1)*qL - vr_hat(3)*qN);
            y_SPH_d_1 = cos(sig/2)*qN + sin(sig/2)*(vr_hat(3)*qM
                - vr_hat(1)*qP - vr_hat(2)*qL);
            z_SPH_d_1 = cos(sig/2)*qP + sin(sig/2)*(vr_hat(1)*qN
                - vr_hat(2)*qM - vr_hat(3)*qL);

% ############################################################
% ############################################################

            % SET TEMPORARY INITAL POSITION
            xt_SPH_d_1(1,1) = xtmp1_SPH_d; % MAKE TEMP INITIALS
               CORRECTION
            yt_SPH_d_1(1,1) = ytmp1_SPH_d;
            zt_SPH_d_1(1,1) = ztmp1_SPH_d;

            % SET TEMPORARY NEXT POSITION
            xt_SPH_d_1(1,2) = x_SPH_d_1;
            yt_SPH_d_1(1,2) = y_SPH_d_1;
            zt_SPH_d_1(1,2) = z_SPH_d_1;

            % MOVE TO NEXT STEP
            mfpt = mfpt + 1;

            % MODIFY NEW INITIAL POSITION FOR NEXT STEP
            xtmp1_SPH_d = x_SPH_d_1;
            ytmp1_SPH_d = y_SPH_d_1;
            ztmp1_SPH_d = z_SPH_d_1;

% ############################################################
% ############################################################
```

```
            % RECORD EACH STEP LOCATION AFTER INITIAL STEP TO
               FINAL STEP
            % ALWAYS HAS tmax + 1 ROWS:
            x_SPH_d_totalrun{1,m}(mfpt+1,1) = x_SPH_d_1;
            y_SPH_d_totalrun{1,m}(mfpt+1,1) = y_SPH_d_1;
            z_SPH_d_totalrun{1,m}(mfpt+1,1) = z_SPH_d_1;

            % TESTING DISTANCE BETWEEN POINT mfpt and mfpt + 1
            % -> DONT NEED THIS FOR PRODUCTION RUN (USE THIS
               WHEN COMPUTING MSD)
            d_S_B3{1,m}(mfpt,1) = 2*R*asin(sqrt(...
              (x_SPH_d_totalrun{1,m}(mfpt+1,1)-x_SPH_d_totalrun
                 {1,m}(mfpt,1))^2 +...
              (y_SPH_d_totalrun{1,m}(mfpt+1,1) -
                 y_SPH_d_totalrun{1,m}(mfpt,1))^2 +...
              (z_SPH_d_totalrun{1,m}(mfpt+1,1) -
                 z_SPH_d_totalrun{1,m}(mfpt,1))^2)/(2*R));

% ############################################################
% ############################################################

            % END SIMULATION IF MAX TIME IS REACHED
            if  (mfpt == tmax) % PLOT TRAJECTORIES UNTIL TMAX IS
                REACHED
                FOR_EX = 1;
                mfpt_totalrun_SPH{1,m}(1,1) = mfpt; % SAVE FINAL
                    TOTAL TIME
                break; % TERMINATE FOR LOOP
            end

% ############################################################

        end % END INCREMENT OVER ALL DISCRETE LEVY WALK

      if FOR_EX == 1
          break; % TERMINATE WHILE LOOP
```

```
      end

    end % END ITERATION OVER ALL TIME-STEPS
end % END ITERATION OVER ALL SAMPLES

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% disp('END MAIN DATA LOOP');
% toc

% CONSTRUCT DATA STRUCTURE TO SAVE
DATA1_SPH_MULTIRUN_MAIN = struct('samples',{samples},'R',{R},'
   mu',{mu},'vel_walk',...
  {vel_walk},'mfpt_totalrun_SPH',{mfpt_totalrun_SPH},'
     x_SPH_d_totalrun',...
  {x_SPH_d_totalrun},'y_SPH_d_totalrun',{y_SPH_d_totalrun},...
  'z_SPH_d_totalrun',{z_SPH_d_totalrun});
save(['LW_DATA1_SPH_MULTIRUN_MAIN',strcat('_MU',num2str(mu)),'
   _V2_',...
  strcat(num2str(tmax/1000),'k_',num2str(samples),'s',strcat('
     _R',num2str(R))),...
  '.mat'],'DATA1_SPH_MULTIRUN_MAIN','-v7.3')

% ############################################################

% % Plot Sphere of radius R and the geodesic path
% % Make Sphere and rescale to radius R
% [X,Y,Z] = sphere;
% xr = R*X;
% yr = R*Y;
% zr = R*Z;
%
% % Plot
% figure('color', 'white','units','normalized','outerposition
   ',[0 0 1 1]); % MAKES IT FULL SCREEN
```

```
% surf(xr,yr,zr,'FaceColor','k','EdgeColor','k','LineStyle
   ',':','FaceAlpha',0.01);
% hold on
% plot3(x_SPH_d_totalrun{1,1},y_SPH_d_totalrun{1,1},...
%   z_SPH_d_totalrun{1,1},'Color','b','LineWidth',2); %
   Euclidean Distance Plot
% xlabel('X', 'FontSize', 20);
% ylabel('Y', 'FontSize', 20);
% zlabel('Z', 'FontSize', 20);
% axis equal;
% axis off
% grid off;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% toc
```

B.2.2 LWALK_IA_SPHERE_MAIN_DATA_MSD.m

This program computes MSD for the Lévy random walk motion in Spherical Space.

```
% Imtiaz Ali, Physics Graduate Student, Univeristy Of
   California, Merced
% Description: Computes geodesic and MSD for Levy walk on
   surface of sphere
%              using the method of quaternion. Sliding window
   method is
%              applied to compute the ensemble-average of time
   -average MSD

% ############################################################
% ############################################################

% Spherical Geodesic Equation:
```

```
% ds = 2*R*arcsin(sqrt(dx^2+dy^2+dz^2)/(2*R))

% ##########################################################
% ##########################################################

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% tic
close all
clear

% LOAD TESTING DATA SET
load('LW_DATA1_SPH_MULTIRUN_MAIN_MU3.4_V2_1k_100s_R100.mat');
samples = DATA1_SPH_MULTIRUN_MAIN.samples;
R = DATA1_SPH_MULTIRUN_MAIN.R;
mu = DATA1_SPH_MULTIRUN_MAIN.mu;
vel_walk = DATA1_SPH_MULTIRUN_MAIN.vel_walk;
mfpt_totalrun_SPH = DATA1_SPH_MULTIRUN_MAIN.mfpt_totalrun_SPH;
x_SPH_d_totalrun = DATA1_SPH_MULTIRUN_MAIN.x_SPH_d_totalrun;
y_SPH_d_totalrun = DATA1_SPH_MULTIRUN_MAIN.y_SPH_d_totalrun;
z_SPH_d_totalrun = DATA1_SPH_MULTIRUN_MAIN.z_SPH_d_totalrun;

% ##########################################################
% ##########################################################

% TIME-AVERAGING
% INITIALIZE CELLS
% MSD FOR ALL SAMPLES -> {1,sample #}(time-step,[MEAN,
   VARIANCE, STD, STEPS])
MSD_SPH_MULTI = cell(1,samples);
% MEAN FOR ALL SAMPLES -> {1,sample #}(time-step,[MEAN,
   VARIANCE, STD, STEPS])
MEAN_SPH_MULTI = cell(1,samples);
```

```
for m = 1:samples % START ITERATION OVER ALL SAMPLES
  % SET INITIALIZATION
  % DATA AS MATRIX (tmax+1,3)
  DATA_SPH = [x_SPH_d_totalrun{1,m},y_SPH_d_totalrun{1,m},
     z_SPH_d_totalrun{1,m}];
  % NUMBER OF DATA POINTS
  num_data_SPH = size(DATA_SPH,1);
  % NUMBER OF TIME-STEPS/DELTA_T VALUES (TMAX)
  delta_T_SPH = floor(num_data_SPH - 1);
  % [MEAN, VARIANCE, STD, STEPS] -> SQUARED DISTANCE DATA
  MSD_SPH_MULTI{1,m} = zeros(delta_T_SPH,4);
  % [MEAN, VARIANCE, STD, STEPS] -> DISTANCE DATA
  MEAN_SPH_MULTI{1,m} = zeros(delta_T_SPH,4);

  % CALCULATE MSD FOR ALL DELTA_T's
  for dt = 1:delta_T_SPH % START ITERATION OVER ALL DELTA_T's
    % START DISTANCE EQUATION -> TMAX ROWS, 3 COLUMNS
    % -> COLM1 = dx, COLM2 = dy, COLM3 = dz
    d_COORDS_SPH = DATA_SPH(1+dt:end,1:2) - DATA_SPH(1:end-dt
       ,1:2);
    % DISTANCE -> COLUMN VECTOR
    % ds^2 = (2*R*asin(sqrt( dx^2+dy^2+dz^2 )/(2*R)))^2, 1
       COLM, SUM OF EACH ROW
    SPH_disp = 2*R*asin((sqrt(sum(d_COORDS_SPH.^2,2))/(2*R)));
    % DISTANCE SQUARED -> COLUMN VECTOR
    SPH_sqrd_disp = SPH_disp.^2;

    % SQUARED DISTANCE
    MSD_SPH_MULTI{1,m}(dt,1) = mean(SPH_sqrd_disp); % AVERAGE
    MSD_SPH_MULTI{1,m}(dt,2) = var(SPH_sqrd_disp,1); %
       VARIANCE
    MSD_SPH_MULTI{1,m}(dt,3) = std(SPH_sqrd_disp,1); %
       STANDARD DEVIATION
    MSD_SPH_MULTI{1,m}(dt,4) = length(SPH_sqrd_disp); % NUMBER
        OF STEPS
```

```
    % DISTANCE
    MEAN_SPH_MULTI{1,m}(dt,1) = mean(SPH_disp); % AVERAGE
    MEAN_SPH_MULTI{1,m}(dt,2) = var(SPH_disp,1); % VARIANCE
    MEAN_SPH_MULTI{1,m}(dt,3) = std(SPH_disp,1); % STANDARD
       DEVIATION
    MEAN_SPH_MULTI{1,m}(dt,4) = length(SPH_disp); % NUMBER OF
       STEPS
  end % END ITERATION OVER ALL DELTA_T's
end % END ITERATION OVER ALL SAMPLES

% SAVE TIME-AVERAGED MSD
% (VERY LARGE FILE, DON'T NEED IT FOR ANYTHING ELSE EXCEPT FOR
    ENSEMBLE AVERAGE)
DATA2_SPH_MULTIRUN_MAIN = struct('samples',{samples},'R',{R},'
   mu',{mu},...
  'mfpt_totalrun_SPH',{mfpt_totalrun_SPH},...
  'MSD_SPH_MULTI',{MSD_SPH_MULTI},'MEAN_SPH_MULTI',{
     MEAN_SPH_MULTI});
save(['LW_DATA2_SPH_MULTIRUN_MAIN',strcat('_MU',num2str(mu)),'
   _V2_',...
  strcat(num2str(mfpt_totalrun_SPH{1,1}/1000),'k_',num2str(
     samples),...
  's',strcat('_R',num2str(R))),'.mat'],'
     DATA2_SPH_MULTIRUN_MAIN','-v7.3');

% ############################################################
% ############################################################

% ENSEMBLE AVERAGE OF TIME-AVERAGED
max_time_SPH = max([mfpt_totalrun_SPH{1,:}]);
% MSD OF ENTIRE RUNS -> (time-step,[MEAN, VARIANCE, STD, STEPS
   ])
MSD_SPH_MULTI_AVG = zeros(max_time_SPH,4);
% MEAN OF ENTIRE RUNS -> (time-step,[MEAN, VARIANCE, STD,
   STEPS])
MEAN_SPH_MULTI_AVG = zeros(max_time_SPH,4);
```

```
for i = 1:max_time_SPH % START ITERATION OVER ALL TIME-STEPS
  % SPHERICAL ALL SAMPLES AVERAGED
  idx_SPH = ~cellfun('isempty',MSD_SPH_MULTI); % ONLY INCLUDE
     ELEMENT WITH VALUES
  SPH_MSD_ALL_SAMPLES = cellfun(@(v)v(i),MSD_SPH_MULTI(idx_SPH
     ));
  % REMOVE ALL NAN'S FROM VECTOR
  SPH_MSD_ALL_SAMPLES = SPH_MSD_ALL_SAMPLES(~isnan(
     SPH_MSD_ALL_SAMPLES));

  idx_SPH_MEAN = ~cellfun('isempty',MEAN_SPH_MULTI); % ONLY
     INCLUDE ELEMENT WITH VALUES
  SPH_MEAN_ALL_SAMPLES = cellfun(@(v)v(i),MEAN_SPH_MULTI(
     idx_SPH_MEAN));
  % REMOVE ALL NAN'S FROM VECTOR
  SPH_MEAN_ALL_SAMPLES = SPH_MEAN_ALL_SAMPLES(~isnan(
     SPH_MEAN_ALL_SAMPLES));

  % SQUARED DISTANCE AVG
  MSD_SPH_MULTI_AVG(i,1) = mean(SPH_MSD_ALL_SAMPLES); %
     AVERAGE
  MSD_SPH_MULTI_AVG(i,2) = var(SPH_MSD_ALL_SAMPLES,1); %
     VARIANCE
  MSD_SPH_MULTI_AVG(i,3) = std(SPH_MSD_ALL_SAMPLES,1); %
     STANDARD DEVIATION
  MSD_SPH_MULTI_AVG(i,4) = max_time_SPH + 1 - i; % NUMBER OF
     STEPS

  % DISTANCE AVG
  MEAN_SPH_MULTI_AVG(i,1) = mean(SPH_MEAN_ALL_SAMPLES); %
     AVERAGE
  MEAN_SPH_MULTI_AVG(i,2) = var(SPH_MEAN_ALL_SAMPLES,1); %
     VARIANCE
  MEAN_SPH_MULTI_AVG(i,3) = std(SPH_MEAN_ALL_SAMPLES,1); %
     STANDARD DEVIATION
```

```
  MEAN_SPH_MULTI_AVG(i,4) = max_time_SPH + 1 - i; % NUMBER OF
     STEPS
end % END ITERATION OVER ALL TIME-STEPS

% SAVE ENSEMBLE AVERAGE OF TIME-AVERAGED MSD
DATA3_SPH_MULTIRUN_MAIN = struct('samples',{samples},'R',{R},'
   mu',{mu},...
  'mfpt_totalrun_SPH',{mfpt_totalrun_SPH},...
  'MSD_SPH_MULTI_AVG',{MSD_SPH_MULTI_AVG},'MEAN_SPH_MULTI_AVG
     ',{MEAN_SPH_MULTI_AVG});
save(['LW_DATA3_SPH_MULTIRUN_MAIN',strcat('_MU',num2str(mu)),'
   _V2_',...
  strcat(num2str(mfpt_totalrun_SPH{1,1}/1000),'k_',...
  num2str(samples),'s',strcat('_R',num2str(R))),'.mat'],'
     DATA3_SPH_MULTIRUN_MAIN','-v7.3');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% toc
```

B.3 Intracellular Transport on Dynamic Actin Networks Programs

B.3.1 simTransMainMSD_FPTD_IA_ADV_V1_Dynamic.c

This is the main program and is used to simulate anomalous transport with dynamic filament networks. It produces files to perform MSD and MFPT analysis.

```
// simTransMainMSD_FPTD_IA_ADV_V1_Dynamic.c: Imtiaz Ali UC
   MERCED
// Description: A program that does anomalous RW of cargo on a
    network of microtubules that can shrink or grow.
//              It calculates MSD, MFPT, FPTD and FLUX
// Compiler: GCC 8.1
```

```
// Last modified: 05/05/2021
// How to run: gcc simTransMainMSD_FPTD_IA_ADV_V1_Dynamic.c
   Net_Setup.c Net_Distances_MINMAX.c Net_Shrink_Grow.c
//             -Ofast -o test1 -lm -std=c99 -m64

#include <stdio.h> // NULL
#include <math.h>  // fabs(), -> Need std=c99 compiler for [
   isnan(), isinf()]
#include <time.h>  // time()
#include <stdlib.h> // rand(), RAND_MAX max value returned by
   rand(), srand()
#include <string.h>

// Include function headers for simulation
#include "Net_Setup.h"
#include "Net_Distances_MINMAX.h"
#include "Net_Shrink_Grow.h"

#define M_PI 3.14159265358979323846

// (Eukaryotic cells are 10's of microns), (Bacteria cells are
    about 1 micron), (Plant cells are 100's of microns)
// Data for MT and Actin gathered from:
// 1. (Craig, E. M. (2018). Model for coordination of
   microtubule and actin dynamics in growth cone turning.
   Frontiers in cellular neuroscience, 12, 394.)
// 2. (Burnette, D. T., Schaefer, A. W., Ji, L., Danuser, G.,
   & Forscher, P. (2007). Filopodial actin bundles are not
   necessary for microtubule advance into the peripheral
   domain of Aplysia neuronal growth cones. Nature cell
   biology, 9(12), 1360.)
// 3. (Mitchison, T., & Kirschner, M. (1988). Cytoskeletal
   dynamics and nerve growth. Neuron, 1(9), 761-772.)
```

```
// 4. (Sarkar, Apurba, Heiko Rieger, and Raja Paul. "Search
   and capture efficiency of dynamic microtubules for
   centrosome relocation during IS formation." Biophysical
   journal 116.11 (2019): 2079-2091.)
float DT = 1.0; // Global filament time-step [seconds] -> The
   same for all particles and filaments
#define V_G_MT 0.0 // (2.) Microtubule Growth Rate [microns/
   sec] 0.1 ->
#define V_S_MT 0.0 // (2.) Microtubule Shrink Rate [microns/
   sec] 0.16 ->
#define V_G_ACTIN 0.60 // (3.) Actin Growth Rate [microns/sec]
    0.083 ->
#define V_S_ACTIN 0.60 // (3.) Actin Shrink Rate [microns/sec]
    0.233 ->
#define f_MT_Res 0.0 // (2.) Microtubule Rescue Frequency [1/
   seconds] 0.0298 -> Change to 0 for no Rescue
#define f_MT_Cat 0.0 // (2.) Microtubule Catastrophy Frequency
    [1/seconds] 0.0102 -> Change to 0 for no Catastrophy
#define f_MT_PAUSE 0.00 // Microtubule Pause Frequency [1/
   seconds] -> Change to >= 1/DT for static network
#define f_ACTIN_Res 0.0 // (2.) ACTIN Rescue Frequency [1/
   seconds] 0.0298 -> Change to 0 for no Rescue
#define f_ACTIN_Cat 0.0 // (2.) ACTIN Catastrophy Frequency
   [1/seconds] 0.0102 -> Change to 0 for no Catastrophy
#define f_ACTIN_PAUSE_BEG 0.0 // Actin Pause Frequency
   Beginning Side [1/seconds] -> Change to >= 1/DT for static
   network
#define f_ACTIN_PAUSE_END 0.0 // Actin Pause Frequency Ending
   Side [1/seconds] -> Change to >= 1/DT for static network

// Actin-f molecular weight = 536 g/mol

int main() {

 // Start system total run time calculation
 double t1 = clock();
```

```
// Set cell and system variables
float outer = 15.0; // Radius of cell (OUTER) (um) ->
   Originally 10.0 -> MUST BE LARGER THAN INNER -> outer*2 =
   diameter
float inner = 5.0; // Radius of nucleus (INNER) (um) =
   microns
int timeIntMax = 500; // Max timesteps -> Originally 20000
int t_lag = 0; // Number of time step until filament reaches
   steady-state for cargo to start moving
int numCargs = 1; // Number of cargos -> Originally 10000
int numNets = 1; // Number of networks
srand(time(NULL)); // Seed, seconds since Jan 01 1970
int production_run = 0; // Save production run data -> False
   = 0. True = 1

// Cargo
// Max x and y values -> Same coordinate system as
   probability evolution system
float xmax = (outer * 2) + 1, ymax = (outer * 2) + 1; // (um)
float xCellCent = xmax / 2, yCellCent = ymax / 2;  // Center
   of cell (um)
float cRad = 0.1; // Cargo radius (um)
float D = 0.051; // Diffusion constant -> (um)^2/s
float v = 1.0; // Cargo ballistic speed -> (um/s) ->
   Kinesin_v = 0.1 to 0.8 (um/s), Myosin_v = 0.2 to 0.5 (um/s
   )
float distStep = 0.1; // Diffusion distance (um)
float distStep_2 = distStep; // Cargo step length on filament
    -> (um) -> Distance step size (will vary) -> Originally
   0.1
float dtReg = (pow(distStep,2)/(4*D)); // Normal diffusion
   time -> 0.049 seconds -> rescale to this value
float dtBal = (distStep_2/v); // Ballistic motion time -> 0.1
    seconds
```

```
float probOn, probOff; // Randomly pick if cargo attached/
   detached to/from filament
// Cargo On and Off rates -> For cargo falling OFF while on
   filament or cargo attaching ON while off filament
float kOn = 5.00, kOff = 0.5, kOn_2, kOff_2; // kOn
   originally 5.0, kOff originally 0.0 [1/seconds]
float switchProb, switchProb_2; // Used in simulation (for
   loop)
float switchProbmin = 0.00; // Cargo min filament switching
   probability
float switchProbmax = 1.00; // Cargo max filament switching
   probability
float switchProb_V[] = {switchProbmin}; //{switchProbmin
   ,0.10,0.20,0.30,0.40,0.50,0.60,0.70,0.80,0.90,
   switchProbmax}; // Cargo swotch prob -> array
size_t sc_V = sizeof(switchProb_V)/sizeof(float); // Array
   length -> switchProb_V
float scale_dec = 1000.0; // Used for rounding purposes ->
   This numerical issue is not solvable, most you can do is
   minimize the error
dtReg = floorf(dtReg*scale_dec)/scale_dec;

// Re-scale ballistic motion time and step length
dtBal = floorf(dtReg*scale_dec)/scale_dec; // Ballistic
   motion time is equal to Regular diffusion time
distStep_2 = v*dtBal; // Ballistic motion step length is re-
   scaled to equal new time
DT = dtReg; // Re-Initialize global variable to respective
   time-step value
float adj_FLAG; // Cargo adjustment flag when attaching or
   switching to filament, FALSE = 0, TRUE = 1

// Time and counter used for proper re-scaled time-step
   multiplier calculation
float time1f, time2f, t_Carg, t_prev;
int t_count;
```

```
// Network setup
// Set current number of filaments and filament length
int numFils; // Used in simulation (for loop)
int minFils = 15; // Min number of filament in network ->
   Originally 100
int maxFils = 100; // Max number of filament in network ->
   Originally 1500
float filLength; // Used in simulation (for loop)
float minLength = 5.0; // Min filament length -> Originally
   5.0 (um)
float maxLength = 8.0; // Max filament length -> Originally
   15.0 (um)
float lenFils_V[] = {minLength}; //{minLength,2.0,3.0,4.0,
   maxLength}; // Filament lengths in network -> array
int numFils_V[] = {minFils}; // Number of filaments in
   network -> array
size_t nf_V = sizeof(numFils_V)/sizeof(int); // Array length
   -> numFils_V
size_t lf_V = sizeof(lenFils_V)/sizeof(float); // Array
   length -> lenFils_V

// Filament dynamic variables
// Random number, fil polarity and distribution, filament
   type and side, motor type and filament growth flag
float rand_m; // Randomly pick if filament will be modified
float rand_SG; // Randomly pick if filament will shrink or
   grow
int fil_pol = 0; // Filament polarity -> Negative = -1, RNG =
    0, Positive = 1
int fil_radial = 0; // Filament radially distributed -> True
   = 1, False = 0
int filament_type = 0; // Filament Type -> Actin = 0,
   Microtubule = 1
int Actin_side = -1; // No side = -1, Beginning side = 0,
   Ending side = 1
```

```
int motor_type = 1; // Motor Type -> Dynein = 0, Kinesin/
   Myosin = 1
int G_true, G_true_2; // Growth boolean-> N/A = -1, False =
   0, True = 1
int Treadmill_Actin = 1; // Actin tredmill boolean-> False =
   0, True = 1
float cargo_direc = 1.0; // Cargo direction on filament,
   Dynein = -1, Kinesin/Myosin = 1, DEFAULT = 1
if (motor_type == 0) {
 cargo_direc = -1.0; // Fix for dynein
}

// Set probability threshold for filament modification and
   shrink/grow
// Microtubule
float mod_prob_M_END = f_MT_PAUSE*DT; // Probability to not
   modify filament
float sg_prob_M_Cat = 0.0000; // f_MT_Cat*DT; // Probability
   to shrink -> sg_prob_M_Cat = CAT_rate * time
float sg_prob_M_Res_V[] = {0.0000}; // f_MT_Res*DT;
   Probability to grow -> sg_prob_M_Res = RES_rate * time
float sg_prob_M_Res; // Used in simulation (for loop)
size_t prob_M_Res_V = sizeof(sg_prob_M_Res_V)/sizeof(float);
   // Array length -> sg_prob_M_Res_V
// Actin
float mod_prob_A_BEG = f_ACTIN_PAUSE_BEG*DT; // Probability
   to not modify Beginning side of filament
float mod_prob_A_END = f_ACTIN_PAUSE_END*DT; // Probability
   to not modify Ending side of filament
float sg_prob_A_Cat = sg_prob_M_Cat; // f_ACTIN_Cat*DT; //
   Probability to shrink -> sg_prob_A_Cat = CAT_rate * time
float sg_prob_A_Res_V[prob_M_Res_V]; // f_ACTIN_Res*DT;
   Probability to grow -> sg_prob_A_Res = RES_rate * time
for (int xxxx = 0; xxxx < prob_M_Res_V; xxxx++) {
 sg_prob_A_Res_V[xxxx] = sg_prob_M_Res_V[xxxx];
}
```

```
float sg_prob_A_Res; // Used in simulation (for loop)
size_t prob_A_Res_V = sizeof(sg_prob_A_Res_V)/sizeof(float);
   // Array length -> sg_prob_M_Res_V
float lr_current; // Actin current length
float r1_current, r2_current, r1_min, r2_max; // Used to find
    inner and outer filament parts to see if at cell boundary

// Filament min length, max length, Current filament length
float fil_buffer = 0.2; // Buffer region (microns) -> minimum
    distance filament needs to be away from boundaries
float min_fil_length = 1.0; // Smallest filament length (
   microns) -> Filament never disappears
float max_fil_length = 2*sqrt(pow(outer,2) - pow(inner,2));
   // Largest filament length (microns) -> (Max Cord Length)
   = 2*sqrt(or^2-ir^2)
float Cur_fil_len; // Place holder to check if filament being
    modified is within min and max range

// Anomalous or Regular diffusion
int REG = 1; // Choose regular or anomalous diffusion ->
   Regular -> REG = 1, Anomalous -> REG = 0
int INS = 0; // Choose to model insulin, True = 1, cargos
   have different start distribution, assumes outer = 10.0
   and inner = 5.0
float randNum; // For anomalous diffusion rng
int anom_FLAG; // Flag to recalculate anomalous diff dt if
   INF or NAN occurs, True=1, False=0
float gamma; // gamma is the alpha in anomalous MSD paper by
   Bryan Maelfeyt
if (REG == 1) {
 gamma = 1.0; // Normal diffusion
} else {
 gamma = 0.8; // Subdiffusive range (0.2,0.4,0.6,0.8,1.0)
}

// Declaration of current filament and cargo variables
```

```
float minx1x2, maxx1x2, miny1y2, maxy1y2;
float rc, theta, xc, yc, d1, d2, d;
float initial_x, initial_y, initial_t;
float r1, r2, alpha, p, x1, x2, y1, y2, diff;
float phi, beta;
float xcNew, ycNew, rcNew;
int delT; // Used for storing cargo and network data if max
   time is not reached

// Declare time varaibles, counter and iterators
float t, tOn, tOff, dt; // Current cargo total time in
   simulation (seconds)
float stepNum_time; // Physical time cargo has left cell
int timeInt, stepNum, stepNum_2; // Time index | Time index
   cargo has left cell | Time index for saving cargo data
int currentCargo;
// Flag for cargo OFF, ON network, simulation max time/point
   of destination and filament on
int OFF, ON, STOP, STOP_2, SWITCHED, m, currentm;
float CargoInfo[numCargs*numNets][12]; // Info of each cargo
   -> cOFF_front, cOFF_back, ckOFF, ckON, ckON_tot, cSwitch,
   cSwitch_tot, tOn, tOff, t, ckOFF_tot, cOFF_end
float cOFF_front, cOFF_back, ckOFF, ckON, ckON_tot, cSwitch,
   cSwitch_tot, ckOFF_tot, cOFF_end; // # time off front, #
   time off back, # kOff, # kOn, # total kOn chance, # switch
   , # total switch chance, # total off chance, # off ends
float Cargo_prev_pos[2]; // Cargo previous time position [x,y
   ]
float Current_Fil_prev_pos[4]; // Current filament cargo is
   on, if applicable, positions [x1,x2,y1,y2]
float Current_Cargo_exterior_FLAG; // Cargo FLAG to make sure
    current position is outside of filament bounds, False =
   0.0, True = 1.0
float Prev_Cargo_exterior_interior_FLAG; // Cargo FLAG to
   make sure previous position is inside of filament bounds,
   False = 0.0, True = 1.0
```

```
float epsi_0 = 1e-5; // Difference threshold tolerance
   (0.000100)
float Fil_ends_FLAG; // Cargo attachment to filament ends or
   perpedicular flag => Ignore = -1.0, Endpoints = 0.0,
   Perpendicular = 1.0
float xc_temp, yc_temp; // Current or previous cargo position
    to previous filament location

// Used to calculate cargo MSD, average time spent on the
   network, variations in MSD at 10s and 100s
float msdArray[timeIntMax][3]; // row = 20000, col = 3 => [
   timestep][0] = Squared Distance, [timestep][1] = counter
   for ensemble, [timestep][2] = time [seconds]
float fracTimeOn[numCargs*numNets];
float msd10[numNets], msd100[numNets]; // Array of number of
   elements = number of networks
float sd, msdCurrent10, msdCurrent100;
float msdSum10, msdSum100;
float av10, av100, var10, var100;
float stdev10, stdev100; // If only 1 network, std and var
   will be zero

// Used in FPTD calculations
int binSize = 1;
float FPTD[timeIntMax];
int fluxOut10[numNets], fluxOut100[numNets]; // Flux is the
   number of cargos that has left the cell
int count10, count100;
float fluxSum10, fluxSum100;
float fluxvar10, fluxvar100;
float fluxav10, fluxav100;
float fluxstdev10, fluxstdev100; // If only 1 network, std
   and var will be zero

// Used in MFPT calculations
float cargoFPTs[numCargs];
```

```
float cargoMFPTs[numNets], cargoFPTstdev[numNets];
float fptSum, fptVar;
float mfptSum, mfptVar, stdevSum;
float totMFPT, totStdev, avgStdev; // If only 1 network, std
   and var will be zero

// Used to calculate average network filament length
float avg_fl; // Used to calculate filament length for mean
   filament length of network
float avg_fil_net[timeIntMax];

/* ########################################### */
/* ########################################### */
// Start simulation

// Algorithm
// CASE_A: Setup network, Random walk of cargo on all
   networks
// CASE_A_1: Initial Network Setup
// CASE_A_2: Start random walk of each cargo on current
   network
// CASE_A_3: Start cargos movement
// CASE_A_4: Start MSD calculations
// CASE_A_5A: Modify filament -> Microtubule ONLY (Shrink/
   Grow)
// CASE_A_5B: Modify filament -> Actin ONLY (Shrink/Grow)
// CASE_A_6A: Check if cargo is ON or OFF the filament
   network
// CASE_A_6B: Calculate networks average filament length
// CASE_A_7: Save current network/cargo data, record msd10/
   msd100, fraction time on network, Start FPTD calculations
// CASE_A_8: Calculate msd10/msd100 for all cargo in network,
    Start MFPT calculations for network
// CASE_B_1: Do MSD analysis
// CASE_B_2: Do MFPT, FPT and Flux analysis
```

```
// Start loop over all filament rescue probability
for (int FresP = 0; FresP < prob_M_Res_V; FresP++) {

 // Filament rescue probability
 sg_prob_M_Res = sg_prob_M_Res_V[FresP];
 sg_prob_A_Res = sg_prob_A_Res_V[FresP];

 // Start loop over all cargo switching probability
 for (int sc = 0; sc < sc_V; sc++) {

  // Cargo switching probability
  switchProb = switchProb_V[sc];

  // Start loop over all filament lengths
  for (int lf = 0; lf < lf_V; lf++) {

   // Filament length
   filLength = lenFils_V[lf];

   // Start loop over all number of filaments
   for (int nf = 0; nf < nf_V; nf++) {

    // Number of filaments
    numFils = numFils_V[nf]; // Originally numFils = minFils;

    // Need to redeclare network array for every new number
       of filament simulations
    // Redeclare filament network arrays -> INITIALIZATION
       ARRAY
    float filNet[numFils][4];  // columns -> 0=r1, 1=theta,
       2=alpha, 3=p
    float filEnds[numFils][4]; // columns -> 0=x1, 1=x2, 2=y1
       , 3=y2

    // Redecalre filament network arrays > PLACEHOLDER
```

```
float filNet_M[numFils][4];  // columns -> 0=r1, 1=theta,
    2=alpha, 3=p
float filEnds_M[numFils][4]; // columns -> 0=x1, 1=x2, 2=
   y1, 3=y2

// Declare actin treadmilling boolean array
int ACT_treadmill_access[numFils]; // N/A = -1, False =
   0, True = 1

// Initialize MSD array and FPTD back to zero
for (int ii = 0; ii < timeIntMax; ii++) {
 msdArray[ii][0] = 0.0; // Incremental squared
    displacment to respective time-interval = MSD
 msdArray[ii][1] = 0.0; // Incremental time by 1.0 to
    respective time-interval = N
 msdArray[ii][2] = 0.0; // Time interval = delta_T
 FPTD[ii] = 0.0; // FPTD
}

// Initialize fracTimeOn back to 0.0 and CargoInfo
for (int ii = 0; ii < numCargs*numNets; ii++) {
 fracTimeOn[ii] = 0.0; // Time cargo is on filament for
    all network
 for (int jj = 0; jj < 12; jj++) {
  CargoInfo[ii][jj] = 0.0; // Info of each cargo ->
     cOFF_front, cOFF_back, ckOFF, ckON, ckON_tot,
     cSwitch, cSwitch_tot, tOn, tOff, t, ckOFF_tot,
     cOFF_end
 }
}
currentCargo = 0; // Range is 0 to (numCargs*numNets - 1)

// Initialize MSD and Flux variation at 10s and 100s
for (int ii = 0; ii < numNets; ii++) {
 msd10[ii] = 0.0; // MSD at 10 seconds
 msd100[ii] = 0.0; // MSD at 100 seconds
```

```
 fluxOut10[ii] = 0; // Flux out at 10 seconds
 fluxOut100[ii] = 0; // Flux out at 100 seconds
}

// Set up end of file name
char sEnd1[1024];
sprintf(sEnd1,"kOn%.2fkOff%.2fnumFil%dfilLen%.2fnumNets%
   dnumCargs%dgamma%.2fSWProb%.2f.txt",kOn,kOff,numFils,
   filLength,numNets,numCargs,gamma,switchProb);

/* ############################################ */
/* ################## CASE_A ################## */
/* ############################################ */

/* ############################################ */
// CASE_A: Setup network, Random walk of cargo on all
   networks

// Start for loop over all number of networks -> Lay down
    different networks
for (int currentNet = 0; currentNet < numNets; currentNet
   ++) {

 /* ############################################ */
 /* ################ CASE_A_1 ################## */
 /* ############################################ */
 // CASE_A_1: Initial Network Setup

 // Start for loop to initialize current networks
    filament setup
 for (int j = 0; j < numFils; j++) {
  // Set network filament segment positions, respective
     angles, polarity > filNet[numFils][4], filEnds[
     numFils][4]
```

```
 Net_Setup(j, 4, filNet, filEnds, outer, inner,
    filLength, xCellCent, yCellCent, numFils, fil_radial
    , fil_pol, fil_buffer);
}

// Initialize cargo FPT array to zero
for (int i = 0; i < numCargs; i++) {
 cargoFPTs[i] = 0.0;
}
// FPTD, Initialize counter for number of cargos that
   have left the cell
count10 = 0;
count100 = 0;

// Intialize network average filament length to zero
for (int k = 0; k < timeIntMax; k++) {
 avg_fil_net[k] = 0.0;
}

/* ############################################ */
/* ################ CASE_A_2 ################ */
/* ############################################ */
// CASE_A_2: Start random walk of each cargo on current
   network

// Start for loop over all cargos in current network ->
   Start movement of cargos
for (int currentCarg = 0; currentCarg < numCargs;
   currentCarg++) {
 // Initialize network and cargo position -> (The data
    we save)

 // Redecalre filament network arrays > FOR SHRINKING or
     GROWING
 // Malloc needs to be applied -> valgrind helps detect
    memory leaks (occurs if we dont de-allocate memory)
```

```
// Comment out if you want to do production run
/* ---------------------------------------- */
float ***filNet_M2 = (float ***)malloc(timeIntMax*
   sizeof(float**));
float ***filEnds_M2 = (float ***)malloc(timeIntMax*
   sizeof(float**));
for (int ii = 0; ii < timeIntMax; ii++) {
 filNet_M2[ii] = (float **)malloc(numFils*sizeof(float
    *));
 filEnds_M2[ii] = (float **)malloc(numFils*sizeof(float
    *));
 for (int jj = 0; jj < numFils; jj++) {
  filNet_M2[ii][jj] = (float *)malloc(4*sizeof(float));
  filEnds_M2[ii][jj] = (float *)malloc(4*sizeof(float))
     ;
 }
}
/* ---------------------------------------- */

// Cargo array => Dynamically Allocate
// Comment out if you want to do production run
/* ---------------------------------------- */
float **Cargo_pos = (float **)malloc(timeIntMax*sizeof(
   float**)); // For cargo array
for (int ii = 0; ii < timeIntMax; ii++) {
 // Cargo
 Cargo_pos[ii] = (float *)malloc(2*sizeof(float));
}
/* ---------------------------------------- */

// // Check to see if heap memory was not assigned
// if ((filNet_M2 == NULL) || (filEnds_M2 == NULL) || (
   Cargo_pos == NULL)) {
//   printf("Out of memory",stderr);
//   exit(0);
// }
```

```
// Every cargo realizes the same initial network
   configuration
// Re-Initialize place holder array for current cargo
   and initial time
for (int iii = 0; iii < numFils; iii++) {
 for (int jjj = 0; jjj < 4; jjj++) {
  // Re-Initialize
  filNet_M[iii][jjj] = filNet[iii][jjj];
  filEnds_M[iii][jjj] = filEnds[iii][jjj];

  // Save initial filament network to arry we will save
      to
  // Comment out if you want to do production run
  /* ---------------------------------------- */
  filNet_M2[0][iii][jjj] = filNet[iii][jjj];
  filEnds_M2[0][iii][jjj] = filEnds[iii][jjj];
  /* ---------------------------------------- */
 }
} // End Re-Initializing place holder array

// Declare and Initialize all Catastrophy Flag to FALSE
    -> Default, we start off growing
int Catas_FLAG_MT[numFils]; // Microtubule Flag for
   Catastrophe, True = 1, False = 0
int Catas_FLAG_A_BEG[numFils]; // Actin Beginning Flag
   for Catastrophe, True = 1, False = 0
int Catas_FLAG_A_END[numFils]; // Actin Ending Flag for
    Catastrophe, True = 1, False = 0
for (int ff = 0; ff < numFils; ff++) {
 Catas_FLAG_MT[ff] = 0; // False
 Catas_FLAG_A_BEG[ff] = 0; // False
 Catas_FLAG_A_END[ff] = 0; // False

 // Initialize actin treadmilling boolean array
 if (Treadmill_Actin == 1) {
```

```
  ACT_treadmill_access[ff] = -1; // N/A = -1, False =
     0, True = 1
 }
}

// Set network length counter to zero
avg_fl = 0.0;
avg_fil_net[0] = filLength;

// Set all time to zero
t = 0.0; // Current cargo total time in simulation (
   seconds)
tOn = 0.0; // Current cargo total time ballistic motion
    on filament (seconds)
tOff = 0.0; // Current cargo total time anomalous
   diffusion off filament (seconds)
stepNum = 0; // Set step cargo has left cell back to
   zero
stepNum_time = 0.0; // Set cargo physical it has left
   cell back to zero (seconds)
dt = 0.0; // Set current cargo's new physical time to
   zero -> Size of time step (seconds)
timeInt = 0; // Set index of reference time to zero
t_count = 0; // Counter for proper time scale
   multiplier, set back to zero
t_prev = 0.0; // Previous step time
t_Carg = 0.0; // Total current time check

// Flag for cargo OFF, ON network, simulation max time/
   point of destination and filament on
OFF = 1; // Cargo always starts off the network
ON = 0;
STOP = 0; // Cell outer membrane not reached yet
STOP_2 = 0; // Max time not reached yet
SWITCHED = 0; // Cargo filament switching flag
m = 0; // Tracker to go through all filaments
```

```
currentm = -1; // Used to keep track of which filament
   currently on -> Not attached to any filament
cOFF_front = 0.0; // Counter falling off front of
   filament
cOFF_back = 0.0; // Counter falling off back of
   filament
ckOFF = 0.0; // Counter kOff executed -> Cargo fell off
    filament (Not the end points meaning walking off)
ckON = 0.0; // Counter kOn executed -> Cargo attached
   to filament
ckON_tot = 0.0; // Counter kOn executed -> Chance of
   cargo attaching to filament
cSwitch = 0.0; // Counter switching to different
   filament
cSwitch_tot = 0.0; // Counter of total chances of
   switching to different filaments
ckOFF_tot = 0.0; // Counter kOFF executed -> Chance of
   cargo falling of filament
cOFF_end = 0.0; // Counter falling off from the ends of
    filament
adj_FLAG = 0.0; // Initialize cargo adjustment flag
   when attaching or switching to filament to FALSE = 0
Fil_ends_FLAG = -1.0; // Initialize cargo attachment to
    filament ends or perpedicular flag => Default is
   Ignore = -1.0

// Cargo position -> (Radial and Angular)
beta = (2*M_PI)*((float)rand())/RAND_MAX; // Starting
   angular position of cargo
// Cargo start radial position
if (INS == 1) {
 // If modling insulin, cargos must have different
    start distribution
 rc = 10 - 5 * sqrt(4 - (((float)rand())/RAND_MAX + 3))
    ; // With Insulin
} else {
```

```
 rc = (inner + 0.2) - (0.2)*(((float)rand())/RAND_MAX);
     // No Insulin
}

// Cargo starting x, y values
xc = filEnds[5][0] + (filLength/3 + 0.2*cRad)*cos(
   filNet[5][1]);
yc = filEnds[5][2] + (filLength/3 + 0.2*cRad)*sin(
   filNet[5][1]);
// xc = filEnds[5][0] + (filLength + 0.2*cRad)*cos(
   filNet[5][1] + filNet[5][2]);
// yc = filEnds[5][2] + (filLength + 0.2*cRad)*sin(
   filNet[5][1] + filNet[5][2]);
// xc = xCellCent + (filNet[5][0] - 0.2*cRad)*cos(
   filNet[5][1] + filNet[5][2]);
// yc = yCellCent + (filNet[5][0] - 0.2*cRad)*sin(
   filNet[5][1] + filNet[5][2]);
// xc = xCellCent + (filNet[5][0] - 0.2*cRad)*cos(
   filNet[5][1]);
// yc = yCellCent + (filNet[5][0] - 0.2*cRad)*sin(
   filNet[5][1]);
// xc = xCellCent + rc * cos(beta);
// yc = yCellCent + rc * sin(beta);
Cargo_prev_pos[0] = xc; // Cargo previous time x-pos
   intialization
Cargo_prev_pos[1] = yc; // Cargo previous time y-pos
   intialization

// Re-Initialize current cargo network array
// Comment out if you want to do production run
/* ---------------------------------------- */
Cargo_pos[0][0] = xc; // Cargo intial x-position
Cargo_pos[0][1] = yc; // Cargo intial y-position
/* ---------------------------------------- */

// Keep track of starting x, y values
```

```
initial_x = xc;
initial_y = yc;

/* ########################################### */
/* ########################################### */

// Start for loop over all timsteps -> Start letting
   cargo "walk"
for (int tt = 1; tt < timeIntMax; tt++) {

 // Implement time lag so we reach steady-state of
    filaments
 if (tt < t_lag) {
  // Update positions and times appropriately
  xc = xc;
  yc = yc;
  t = floorf((t)*scale_dec)/scale_dec + floorf((dtReg)*
     scale_dec)/scale_dec;
 }

 // Only simulate cargo if still inside cell and max
    time not reached
 if ((STOP == 0) && (STOP_2 == 0) && (tt >= t_lag)) {

  /* ========================================== */
  /* ========================================== */
  // If OFF, allow possibility of attachment to nearby
     filaments
  if (OFF == 1 && ON == 0) {

   // Compute time-step
   if (REG == 0) {
    randNum = ((float)rand())/RAND_MAX;
    randNum = (1 - pow(((float)timeIntMax),-gamma))*
       randNum;
    dt = pow((-randNum+1),(-1/gamma))*(dtReg);
```

```
 dt = floorf(dt*scale_dec)/scale_dec;
 // Anomalous diff timestep -> dt = ((1-(1-(
    timeIntMax^(-gamma)))*rng)^(-1/gamma))*dtReg

 // Check if dt is INF or NAN
 anom_FLAG = 0; // Start False
 if ((isinf(dt) != 0) || (isnan(dt) != 0)) {
  // Need to recalculate dt
  anom_FLAG = 1;
 }
 // Start to recompute dt until we get non-INF and
    non-NAN
 while (anom_FLAG == 1) {
  randNum = ((float)rand())/RAND_MAX;
  randNum = (1 - pow(((float)timeIntMax),-gamma))*
     randNum;
  dt = pow((-randNum+1),(-1/gamma))*(dtReg);
  dt = floorf(dt*scale_dec)/scale_dec;
  // Check if dt is INF or NAN
  if ((isinf(dt) == 0) && (isnan(dt) == 0)) {
   // Need to recalculate dt
   anom_FLAG = 0;
  }
 } // End while loop to recompute dt
} else {
 dt = floorf(dtReg*scale_dec)/scale_dec; // Normal
    diff timestep -> dtReg = (distStep^2)/(4*D)
}

/* ========================================== */
// Check for nearby filaments and filament endpoints
    -> (Must be within cargo radius)
m = 0; // Restarts for each cargo while it is still
   able to simulate
```

```
while (m < numFils && ON != 1) { // start going
   through all filaments to see if cargo is near a
   filament

 // Calculate perp dist, and cargo distance from
    filament segments endpoints -> d, d1, d2
 Net_Distances(&d, &d1, &d2, xc, yc, m, 4, filEnds_M
    );

 // Calculate filament segments min/max (x,y)
    positions -> minx1x2, maxx1x2, miny1y2, maxy1y2
 Net_MINMAX_1(&minx1x2, &maxx1x2, &miny1y2, &maxy1y2
    , m, 4, filEnds_M);

 // Check if cargo is near a filament end
 if ( (d1 < cRad) || (d2 < cRad) ||
  ( (xc > minx1x2) && (xc < maxx1x2) && (yc >
     miny1y2) && (yc < maxy1y2) && (d < cRad) ) ) {

  // Make proper condition so we have equality in
     probability
  if ((kOn*dt) == 0.0) {
   kOn_2 = -1.0;
  } else {
   kOn_2 = kOn;
  }

  ckON_tot += 1.0; // Cargo chance to attach to
     filaments increment
  probOn = ((float)rand())/RAND_MAX; // Probability
     of attaching to nearby filament
  if (probOn <= (kOn_2*dt)) {
   ckON += 1.0; // Cargo attached increment
   ON = 1; // Current cargo attaches to filament
   theta = filNet_M[m][1];
   alpha = filNet_M[m][2];
```

```
p = filNet_M[m][3];
x1 = filEnds_M[m][0];
x2 = filEnds_M[m][1];
y1 = filEnds_M[m][2];
y2 = filEnds_M[m][3];
currentm = m; // Current filament number

// Set flag
adj_FLAG = 1.0; // Cargo transition from
   Diffusion to Ballistic FLAG
// Save cargo intial position before update
Cargo_prev_pos[0] = xc; // Cargo previous time x-
   pos intialization
Cargo_prev_pos[1] = yc; // Cargo previous time y-
   pos intialization

// Adjust cargo position to be exactly on
   filament (Valid because within cargo radius)
if (d1 < cRad) {
 // Attach to first side
 Fil_ends_FLAG = 0.0;
 xc = x1;
 yc = y1;
 // printf("Attached to Beginning Side, t = %d\n
    ",tt);
} else if (d2 < cRad) {
 // Attach to second side
 Fil_ends_FLAG = 0.0;
 xc = x2;
 yc = y2;
 // printf("Attached to Ending Side, t = %d\n",tt
    );
} else if (d < cRad) {
 // Attach to connecting perpendicular side => (
    Most robust form)
 Fil_ends_FLAG = 1.0;
```

```
      xc = ( pow((x2-x1),2)*xc + (y2-y1)*(x2-x1)*yc -
         (y2-y1)*(x2*y1 - x1*y2) )/(pow((y2-y1),2) +
         pow((x2-x1),2));
      yc = -((y2-y1)/(x1-x2))*xc - ((x2*y1 - x1*y2)/(
         x1-x2));
      // printf("Attached Perpendicular, t = %d\n",tt)
         ;
     }
    } // End condition check to attach to filament
   } // End condition chech to see if nearby filament
   m += 1; // Increment to next filament
  } // End while loop through all filaments
  /* ========================================== */

 } // End condition to allow possibility of attachment
     to nearby filaments
 /* ========================================== */
 /* ========================================== */

 /* ========================================== */
 /* ========================================== */
 // If ON, allow possibility of cargo to detach,
    switch filament or being stuck at filament block
 if (ON == 1 && OFF == 0) {
  dt = floorf(dtBal*scale_dec)/scale_dec; // Ballistic
      motion -> dt = distStep_2 / v

  // Calculate relative filament segments min/max (x,y
     ) position -> minx1x2, maxx1x2, miny1y2, maxy1y2
  Net_MINMAX_2(&minx1x2, &maxx1x2, &miny1y2, &maxy1y2,
      x1, x2, y1, y2);

  // Make proper condition so we have equality in
     probability
  if ((kOff*dt) == 0.0) {
   kOff_2 = -1.0;
```

```
} else {
 kOff_2 = kOff;
}

ckOFF_tot += 1.0; // Cargo chance to fall off
   filament increment
probOff = ((float)rand())/RAND_MAX; // Probability
   of detaching from network filament
/* ========================================== */
// Check if cargo will detach from current filament
// if ( (probOff <= (kOff_2*dt)) || ((xc<minx1x2) ||
    (xc>maxx1x2) || (yc<miny1y2) || (yc>maxy1y2)) )
   {
if ( (probOff <= (kOff_2*dt)) ) {

 // Update flags since cargo has fallen off the
    network
 ON = 0; // Current cargo will not be on current
    filament in next step
 OFF = 1; // Current cargo will fall off current
    filament in next step
 currentm = -1; // Not attached to any filament
 ckOFF += 1.0; // Cargo detached increment
 // printf("Detached koff, t = %d\n",tt);

 // Compute time-step
 if (REG == 0) {
  randNum = ((float)rand())/RAND_MAX;
  randNum = (1 - pow(((float)timeIntMax),-gamma))*
     randNum;
  dt = pow((-randNum+1),(-1/gamma))*(dtReg);
  dt = floorf(dt*scale_dec)/scale_dec;
  // Anomalous diff timestep -> dt = ((1-(1-(
     timeIntMax^(-gamma)))*rng)^(-1/gamma))*dtReg

  // Check if dt is INF or NAN
```

```
  anom_FLAG = 0; // Start False
  if ((isinf(dt) != 0) || (isnan(dt) != 0)) {
   // Need to recalculate dt
   anom_FLAG = 1;
  }
  // Start to recompute dt until we get non-INF and
     non-NAN
  while (anom_FLAG == 1) {
   randNum = ((float)rand())/RAND_MAX;
   randNum = (1 - pow(((float)timeIntMax),-gamma))*
      randNum;
   dt = pow((-randNum+1),(-1/gamma))*(dtReg);
   dt = floorf(dt*scale_dec)/scale_dec;
   // Check if dt is INF or NAN
   if ((isinf(dt) == 0) && (isnan(dt) == 0)) {
    // Need to recalculate dt
    anom_FLAG = 0;
   }
  } // End while loop to recompute dt
 } else {
  dt = floorf(dtReg*scale_dec)/scale_dec; // Normal
     diff timestep -> dtReg = (distStep^2)/(4*D)
 }
} // End check if cargo has fallen off
/* ========================================== */

/* ========================================== */
// If ON, cargo still did not detach -> Check for
   swithcing to nearby filaments
if (ON == 1 && OFF == 0) {

 m = 0; // Cycle through filament number
 SWITCHED = 0; // While loop flag -> Cargo has not
    switched yet
 // Loop through filaments -> If there is a filament
     nearby, allow probability to switch
```

```
while (m < numFils && SWITCHED != 1) {

 // Calculate perp dist, and cargo distance from
    filament segments endpoints -> d, d1, d2
 Net_Distances(&d, &d1, &d2, xc, yc, m, 4,
    filEnds_M);

 // Calculate filament segments min/max (x,y)
    positions -> minx1x2, maxx1x2, miny1y2, maxy1y2
 Net_MINMAX_1(&minx1x2, &maxx1x2, &miny1y2, &
    maxy1y2, m, 4, filEnds_M);

 // Check if cargo will switch to another filament
    if nearby
 if ( ( (d1 < cRad) || (d2 < cRad) ||
   ((xc>minx1x2) && (xc<maxx1x2) && (yc>miny1y2) &&
       (yc<maxy1y2) && (d<cRad)) ) && (currentm !=
      m) ) {

  // Make proper condition so we have equality in
     probability
  if (switchProb == 0.0) {
   switchProb_2 = -1.0;
  } else {
   switchProb_2 = switchProb;
  }

  cSwitch_tot += 1.0; // Cargo chance to switch
     filaments increment
  probOn = ((float)rand())/RAND_MAX; // Probability
      of switching to another filament
  if (probOn <= switchProb_2) {
   // Cargo switches over to another filament
   cSwitch += 1.0; // Cargo switched filaments
      increment
```

```
SWITCHED = 1; // Current cargo switched to
   another filament
theta = filNet_M[m][1];
alpha = filNet_M[m][2];
p = filNet_M[m][3];
x1 = filEnds_M[m][0];
x2 = filEnds_M[m][1];
y1 = filEnds_M[m][2];
y2 = filEnds_M[m][3];
currentm = m; // Update current filament number

// Set flag
adj_FLAG = 1.0; // Cargo transition from
   Diffusion to Ballistic FLAG
// Save cargo intial position before update
Cargo_prev_pos[0] = xc; // Cargo previous time x
   -pos intialization
Cargo_prev_pos[1] = yc; // Cargo previous time y
   -pos intialization

// Adjust cargo position to be exactly on
   filament (Valid because within cargo radius)
if (d1 < cRad) {
 // Attach to first side
 Fil_ends_FLAG = 0.0;
 xc = x1;
 yc = y1;
} else if (d2 < cRad) {
 // Attach to second side
 Fil_ends_FLAG = 0.0;
 xc = x2;
 yc = y2;
} else if (d < cRad) {
 // Attach to connecting perpendicular side => (
    Most robust form)
 Fil_ends_FLAG = 1.0;
```

```
      xc = ( pow((x2-x1),2)*xc + (y2-y1)*(x2-x1)*yc -
          (y2-y1)*(x2*y1 - x1*y2) )/(pow((y2-y1),2) +
          pow((x2-x1),2));
      yc = -((y2-y1)/(x1-x2))*xc - ((x2*y1 - x1*y2)/(
         x1-x2));
     }
    } // End condition check of switching filaments
   } // End condition to check of being in range of
      another filament
   m += 1; // Increment to next filament
  } // End while loop over all filament
 } // End condition to check if switches to nearby
    filaments
 /* ========================================== */

} // End condition, if ON, allow prob of falling off
   current filament or switching filaments
/* ========================================== */
/* ========================================== */

/* ########################################## */
/* ################# CASE_A_3 ############### */
/* ########################################## */
// CASE_A_3: Start cargos movement

// Cargo previous time location (Do this before
   update)
if (adj_FLAG == 0.0) {
 Cargo_prev_pos[0] = xc; // Cargo previous time x-pos
 Cargo_prev_pos[1] = yc; // Cargo previous time y-pos
}

// Now that the cargo is either on or off, allow
   movement make sure that cargo is indeed on a
   filament
if (ON == 1 && OFF == 1) {
```

```
 ON = 1;
 OFF = 0;
 dt = floorf(dtBal*scale_dec)/scale_dec; // Ballistic
     motion -> dt = distStep_2 / v
}

// Random walk OFF filament network -> DIFFUSION
if (OFF == 1) {
 phi = (2*M_PI)*((float)rand())/RAND_MAX; // Pick
    random direction
 // Move in that direction through cytoplasm
 xcNew = xc + (distStep) * cos(phi);
 ycNew = yc + (distStep) * sin(phi);
 tOff = floorf((tOff)*scale_dec)/scale_dec + floorf((
    dt)*scale_dec)/scale_dec; // Update current cargo
     total time off filament
}

// Ballistic motion on filament network -> BALLISITC
if (ON == 1) {
 printf("BALLISTIC, t = %d\n",tt);

 if (adj_FLAG == 1.0) {
  // Transitioned to filament
  xcNew = xc;
  ycNew = yc;
  adj_FLAG = 0.0; // Change back to FALSE
 } else {
  // Move along filament polarity
  xcNew = xc + cargo_direc * p * distStep_2 * cos(
     theta + alpha);
  ycNew = yc + cargo_direc * p * distStep_2 * sin(
     theta + alpha);
 }
```

```
 tOn = floorf((tOn)*scale_dec)/scale_dec + floorf((dt
    )*scale_dec)/scale_dec; // Update current cargo
    total time on filament

 // Current filament cargo is on end positions (Do
    this before update)
 Current_Fil_prev_pos[0] = x1; // Filament x1
    position (use polarity to find if its beginning
    or ending side)
 Current_Fil_prev_pos[1] = x2; // Filament x2
    position (use polarity to find if its beginning
    or ending side)
 Current_Fil_prev_pos[2] = y1; // Filament y1
    position (use polarity to find if its beginning
    or ending side)
 Current_Fil_prev_pos[3] = y2; // Filament y2
    position (use polarity to find if its beginning
    or ending side)
}

// New radial  and angular position of cargo
rcNew = sqrt(pow((xcNew-xCellCent),2)+pow((ycNew-
   yCellCent),2));
beta = atan((ycNew-yCellCent)/(xcNew-xCellCent));
// Check if cargo is inside nucleus -> If cargo
   inside nucleus, move back out to original position
if (rcNew < inner) {
 xcNew = xc;
 ycNew = yc;
} else if (rcNew > outer) {
 // Check if the cargo is outside cell -> If cargo
    outside cell, move back inside to original
    position
 t_Carg = floorf((t)*scale_dec)/scale_dec + floorf((
    dt)*scale_dec)/scale_dec;
 if (t_Carg < ((float)(timeIntMax))*dtReg) {
```

```
    STOP = 1; // For FPTD -> If cargo has left the cell
       , stop cargo movement
   }

   // For MSD calculations, two reflecting boundaries (
      Bounce off the Cell Membrane)
   xcNew = xc;
   ycNew = xc;
  }
  // Update positions and times appropriately
  xc = xcNew;
  yc = ycNew;
  rc = sqrt(pow((xc-xCellCent),2)+pow((yc-yCellCent),2)
     );
  beta = atan((yc-yCellCent)/(xc-xCellCent));
  t = floorf((t)*scale_dec)/scale_dec + floorf((dt)*
     scale_dec)/scale_dec; // Update total simulation
     time of current cargo

 } // End check if cargo in still inside cell membrane
    -> STOP == 0

 /* ############################################ */
 /* ################ CASE_A_4 ################ */
 /* ############################################ */
 // CASE_A_4: Start MSD calculations

 // DUE TO TIME CONSTRAINTS, I COULD NOT FINISH THE
    PROPER TIME SCALING SUCH THAT THE TIME-INTERVAL
    WILL BE IN ACCORDANCE
 // TO THE PHYSICAL TIME MOVED IN THE GIVEN ITERATION,
    THUS THIS PART ONLY WORKS FOR DIFFUSION, NOT SUPER
    OR SUB-DIFFUSION.
 // ANY QUESTIONS ABOUT THIS SHOULD BE DIRECTED TO
    IMTIAZ ALI
```

```
// // Make sure we are at proper dtReg timescale
   multiplier
// time1f = (t/dtReg) - roundf(t/dtReg);
// time2f = (t_prev/dtReg) - roundf(t_prev/dtReg);
// if ((fabs(time1f - time2f)) >= 1.0) {
//  t_count += (int)(roundf(fabs(time1f - time2f)));
// }
// timeInt = ((int)roundf(t/dtReg) + t_count); // Time
    interval -> rescaled wrt dtReg
// t_prev = t; // Update to current time, used in next
    step
timeInt = tt; // Time interval -> rescaled wrt dtReg (
   Method only works for DIFFUSION NOT SUB/SUPER
   DIFFUSION)

// Start MSD storage
if (STOP == 0) {
 // Calculate cargo square distance from initial
    starting position
 sd = pow((xc - initial_x),2) + pow((yc - initial_y)
    ,2);

 if ((int)t == 10) {
  msdCurrent10 = sd;
 } else if ((int)t == 100) {
  msdCurrent100 = sd;
 }

 // Store cargo position
 // Comment out if you want to do production run
 /* ---------------------------------------- */
 Cargo_pos[timeInt][0] = xc; // Cargo x-position
 Cargo_pos[timeInt][1] = yc; // Cargo y-position
 /* ---------------------------------------- */

 if (timeInt < timeIntMax) {
```

```
  // Max simulation time not reached
  msdArray[timeInt][0] += sd;
  msdArray[timeInt][1] += 1.0;
  msdArray[timeInt][2] = ((float)timeInt)*dtReg;
 } else if (timeInt > timeIntMax) {
  // Max simulation time reached
  STOP_2 = 1; // Stop simulation of cargo only -> Keep
      computing for filaments
 }
} // End MSD storage

if (STOP == 1) {
 // Not inside cell membrane
 sd = 0.0;
 if (timeInt < timeIntMax) {
  // Max simulation time not reached
  stepNum = timeInt; // Time step cargo left cell
  stepNum_time = t; // Physical total time (int)roundf
     (t/dtReg)
 }
}

/* ############################################ */
/* ################ CASE_A_5A ################ */
/* ############################################ */
// CASE_A_5A: Modify filament -> Microtubule ONLY

if (filament_type == 1) {
 // Modify all the filaments for simplicity -> Let
    mod_prob_M_END = 0.0
 // Start iteration over all filaments
 for (int kk = 0; kk < numFils; kk++) {

  /* ======================================== */
  // Filament will not grow if length is max and not
     shrink if length is min
```

```
// Current filaments length
Cur_fil_len = sqrt(pow((filEnds_M[kk][1] - filEnds_M
   [kk][0]),2) + pow((filEnds_M[kk][3] - filEnds_M[
   kk][2]),2));

// Choose if we modify filament -> (rand_m >
   mod_prob_M_END means to modify filament)
rand_m = ((float)rand())/RAND_MAX;
if (rand_m < mod_prob_M_END) {
 G_true = -1; // N/A -> Do Nothing
} else {

 // This part works with catastrophy and Rescue
 // Check if catastrophy has occured
 if (Catas_FLAG_MT[kk] == 0) {
  rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
     sg_prob_M_Cat => means to shrink filament (
     CATASTROPHY)
  if ((rand_SG <= sg_prob_M_Cat) && (Cur_fil_len >
     min_fil_length)) {
   Catas_FLAG_MT[kk] = 1; // True -> Catastrophy
      Occured
  }
 } else if (Catas_FLAG_MT[kk] == 1) {
  // Check if rescue has occured
  rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
     sg_prob_M_Res => means to grow filament (RESCUE
     )
  if ((rand_SG <= sg_prob_M_Res) && (Cur_fil_len <
     max_fil_length)) {
   Catas_FLAG_MT[kk] = 0; // False -> Recovery
      Occured => Originally if (Cur_fil_len <=
      min_fil_length)
  }
 }
```

```
    if (Catas_FLAG_MT[kk] == 1) {
     G_true = 0; // False -> Filament will Shrink
    } else if (Catas_FLAG_MT[kk] == 0) {
     G_true = 1; // True -> Filament will Grow
    }
   }
   // Modify filament
   Net_SG_MICROTUBULE(kk, 4, filNet_M, filEnds_M, outer
      , inner, xCellCent, yCellCent, G_true,
    DT, V_G_MT, V_S_MT, min_fil_length, fil_buffer);

   // Store in M2 array
   // Comment out if you want to do production run
   /* ---------------------------------------- */
   for (int qq = 0; qq < 4; qq++) {
    filNet_M2[tt][kk][qq] = filNet_M[kk][qq];
    filEnds_M2[tt][kk][qq] = filEnds_M[kk][qq];
   }
   /* ---------------------------------------- */
   /* ======================================== */

 } // End Modifying all filaments

} // End check of filament type -> MICROTUBULE

/* ######################################### */
/* ################ CASE_A_5B ################ */
/* ######################################### */
// CASE_A_5B: Modify filament -> Actin ONLY

if (filament_type == 0) {
 // Modify all the filaments for simplicity -> Let
    mod_prob_A = 0.0
 // Start iteration over all filaments
 for (int kk = 0; kk < numFils; kk++) {
```

```
/* ########################################### */
// Explicity look at Beginning side

/* ========================================== */
// Current filaments length
Cur_fil_len = sqrt(pow((filEnds_M[kk][1] - filEnds_M
   [kk][0]),2) + pow((filEnds_M[kk][3] - filEnds_M[
   kk][2]),2));

Actin_side = 0;
if (Treadmill_Actin == 0) {
 // No Actin treadmilling
 // Choose if we modify filament -> (rand_m >
    mod_prob_A_BEG means to modify filament)
 rand_m = ((float)rand())/RAND_MAX;
 if (rand_m < mod_prob_A_BEG) {
  G_true = -1; // N/A -> Do Nothing
 } else {

  // This part works with catastrophy and rescue
  // Check if catastrophy has occured
  if (Catas_FLAG_A_BEG[kk] == 0) {
   rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
       sg_prob_A_Cat => means to shrink filament (
      CATASTROPHY)
   if ((rand_SG <= sg_prob_A_Cat) && (Cur_fil_len >
      min_fil_length)) {
    Catas_FLAG_A_BEG[kk] = 1; // True -> Catastrophy
        Occured
   }
  } else if (Catas_FLAG_A_BEG[kk] == 1) {
   // Check if rescue has occured
   rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
       sg_prob_A_Res => means to grow filament (
      RESCUE)
```

```
    if ((rand_SG <= sg_prob_A_Res) && (Cur_fil_len <
       max_fil_length)) {
     Catas_FLAG_A_BEG[kk] = 0; // False -> Recovery
        Occured
    }
   }

   if (Catas_FLAG_A_BEG[kk] == 1) {
    G_true = 0; // False -> Filament will Shrink
   } else if (Catas_FLAG_A_BEG[kk] == 0) {
    G_true = 1; // True -> Filament will Grow
   }
  }
 } else {
  // Check if dynamic for actin is not set for
     treadmilling
  if (ACT_treadmill_access[kk] == -1) {
   // Need to set beginning side to shrink or grow

   // Choose if we modify filament -> (rand_m >
      mod_prob_A_BEG means to modify filament)
   rand_m = ((float)rand())/RAND_MAX;
   if (rand_m < mod_prob_A_BEG) {
    G_true = -1; // N/A -> Do Nothing
   } else {

    // Treadmilling, Beginning end shrinks
    Catas_FLAG_A_BEG[kk] == 1; // True -> Catastrophy
        Occured
    G_true = 0; // False -> Filament will Shrink
   }
   // Adjust actin tredmill flag
   ACT_treadmill_access[kk] = G_true;

  } else {
   // Keep tredmill going
```

```
  G_true = ACT_treadmill_access[kk];
 }
}

// Modify filament
Net_SG_ACTIN(kk, 4, filNet_M, filEnds_M, outer,
   inner, xCellCent, yCellCent, G_true, Actin_side,
 DT, V_G_ACTIN, V_S_ACTIN, min_fil_length,
    fil_buffer);
Actin_side = -1; // Change back to looking at no
   side

// Store in M2 array
// Comment out if you want to do production run
/* ---------------------------------------- */
for (int qq = 0; qq < 4; qq++) {
 filNet_M2[tt][kk][qq] = filNet_M[kk][qq];
 filEnds_M2[tt][kk][qq] = filEnds_M[kk][qq];
}
/* ---------------------------------------- */

/* ######################################## */
// Explicity look at Ending side

// Current filaments length
Cur_fil_len = sqrt(pow((filEnds_M[kk][1] - filEnds_M
   [kk][0]),2) + pow((filEnds_M[kk][3] - filEnds_M[
   kk][2]),2));

Actin_side = 1;
if (Treadmill_Actin == 0) {
 // No Actin treadmilling
 // Choose if we modify filament -> (rand_m >
    mod_prob_A_END means to modify filament)
 rand_m = ((float)rand())/RAND_MAX;
```

```
 if ((rand_m < mod_prob_A_END) || (Cur_fil_len >
    max_fil_length)) {
  G_true = -1; // N/A -> Do Nothing
 } else {

  // This part works with catastrophy and rescue
  // Check if catastrophy has occured
  if (Catas_FLAG_A_END[kk] == 0) {
   rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
       sg_prob_A_Cat => means to shrink filament (
      CATASTROPHY)
   if ((rand_SG <= sg_prob_A_Cat) && (Cur_fil_len >
      min_fil_length)) {
    Catas_FLAG_A_END[kk] = 1; // True -> Catastrophy
        Occured
   }
  } else if (Catas_FLAG_A_END[kk] == 1) {
   // Check if rescue has occured
   rand_SG = ((float)rand())/RAND_MAX; // rand_SG <=
       sg_prob_A_Res => means to grow filament (
      RESCUE)
   if ((rand_SG <= sg_prob_A_Res) && (Cur_fil_len <
      max_fil_length)) {
    Catas_FLAG_A_END[kk] = 0; // False -> Recovery
        Occured
   }
  }

  if (Catas_FLAG_A_END[kk] == 1) {
   G_true = 0; // False -> Filament will Shrink
  } else if (Catas_FLAG_A_END[kk] == 0) {
   G_true = 1; // True -> Filament will Grow
  }
 }
 G_true_2 = G_true;
} else {
```

```
 // Actin treadmilling
 // Choose if we modify filament -> (We want it such
     that if beginning side shrinks, ending side
    will grow and vise versa -> treadmilling)
 if (ACT_treadmill_access[kk] == 1) {
  G_true_2 = 0; // Filament ending side will shrink
 } else if (ACT_treadmill_access[kk] == 0) {
  G_true_2 = 1; // Filament ending side will grow
 } else {
  G_true_2 = ACT_treadmill_access[kk]; // Filament
     ending side will do nothing
 }
}

// Modify filament
Net_SG_ACTIN(kk, 4, filNet_M, filEnds_M, outer,
   inner, xCellCent, yCellCent, G_true_2, Actin_side
   ,
 DT, V_G_ACTIN, V_S_ACTIN, min_fil_length,
    fil_buffer);
Actin_side = -1; // Change back to looking at no
   side

// Check if actin is at minimum length and at
   boundaries -> Need to rearrange if so
// Calculate filaments current length
lr_current = sqrt(pow((filEnds_M[kk][1] - filEnds_M[
   kk][0]),2) + pow((filEnds_M[kk][3] - filEnds_M[kk
   ][2]),2));
r1_current = sqrt(pow((filEnds_M[kk][0] - xCellCent)
   ,2) + pow((filEnds_M[kk][2] - yCellCent),2)); //
   Beginning side
r2_current = sqrt(pow((filEnds_M[kk][1] - xCellCent)
   ,2) + pow((filEnds_M[kk][3] - yCellCent),2)); //
   Ending side
// Find inner and outer regions to boundary
```

```
if (r1_current < r2_current) {
 r1_min = r1_current;
 r2_max = r2_current;
} else {
 r1_min = r2_current;
 r2_max = r1_current;
}
if ( (lr_current < (min_fil_length + 0.1)) ) {
 // New filament arrangement
 Net_Setup(kk, 4, filNet_M, filEnds_M, outer, inner,
     filLength, xCellCent, yCellCent, numFils,
    fil_radial, fil_pol, fil_buffer);
 Catas_FLAG_A_BEG[kk] = 0; // False
 Catas_FLAG_A_END[kk] = 0; // False
 // Reset actin treadmilling  boolean array
 if (Treadmill_Actin == 1) {
  ACT_treadmill_access[kk] = -1; // N/A = -1, False
     = 0, True = 1
 }

 // Check if cargo was on filament
 if (ON == 1) {
  if (currentm == kk) {
   // Filament disappeared
   ckOFF += 1.0; // Cargo detached increment
   ON = 0; // Current cargo is not be on current
      filament
   OFF = 1; // Current cargo is diffusion
   currentm = -1; // Not attached to any filament
  }
 }
} // End check if actin filament needed rearrangment

// Store in M2 array
// Comment out if you want to do production run
/* ---------------------------------------- */
```

```
  for (int qq = 0; qq < 4; qq++) {
   filNet_M2[tt][kk][qq] = filNet_M[kk][qq];
   filEnds_M2[tt][kk][qq] = filEnds_M[kk][qq];
  }
  /* ---------------------------------------- */
  /* ======================================== */

  /* ######################################## */
 } // End Modifying all filaments

} // End check of filament type -> ACTIN

/* ############################################ */
/* ################ CASE_A_6A ################ */
/* ############################################ */
// CASE_A_6A: Check if cargo is ON or OFF the filament
    network after modification

// Only check if cargo is still inside cell membrane
   or max time not reached
if ((STOP == 0) && (STOP_2 == 0)) {

 // If ON, check if cargo fell off modified filament
 if (ON == 1 && OFF == 0) {

  // If cargo falls off of current filament, it will
     be from its endpoints
  m = currentm; // Current filament cargo is on
  // Current filament endpoint positions
  x1 = filEnds_M[m][0];
  x2 = filEnds_M[m][1];
  y1 = filEnds_M[m][2];
  y2 = filEnds_M[m][3];

  // Calculate relative filament segments min/max (x,y
     ) position -> minx1x2, maxx1x2, miny1y2, maxy1y2
```

```
Net_MINMAX_2(&minx1x2, &maxx1x2, &miny1y2, &maxy1y2,
    x1, x2, y1, y2);
if ( (xc<minx1x2) || (xc>maxx1x2) || (yc<miny1y2) ||
    (yc>maxy1y2) ) {

 if (Fil_ends_FLAG == 0.0) {
  // Cargo attached to endpoint (Use current cargo
     position)
  xc_temp = xc;
  yc_temp = yc;
 } else {
  // Cargo attached between endpoints (Use previous
     cargo position)
  xc_temp = Cargo_prev_pos[0];
  yc_temp = Cargo_prev_pos[1];
 }

 // Start check if previous cargo position is on
    filament at its previous location
 // Find current filaments previous max and min
 Net_MINMAX_2(&minx1x2, &maxx1x2, &miny1y2, &maxy1y2
    ,
  Current_Fil_prev_pos[0], Current_Fil_prev_pos[1],
     Current_Fil_prev_pos[2], Current_Fil_prev_pos
     [3]);
 Prev_Cargo_exterior_interior_FLAG = 0.0; // Set to
    False = 0.0, Only change it if the conditions
    are met
 if (miny1y2 == maxy1y2) {
  // Check horizontal case
  if ( (xc_temp >= minx1x2) && (xc_temp <= maxx1x2)
     ) {
   Prev_Cargo_exterior_interior_FLAG = 1.0;
  }
 } else if (minx1x2 == maxx1x2) {
  // Check vertical case
```

```
 if ( (yc_temp >= miny1y2) && (yc_temp <= maxy1y2)
    ) {
  Prev_Cargo_exterior_interior_FLAG = 1.0;
 }
} else if (( (xc_temp >= minx1x2) &&
 (xc_temp <= maxx1x2) && (yc_temp >= miny1y2) && (
    yc_temp <= maxy1y2) )) {
 Prev_Cargo_exterior_interior_FLAG = 1.0;
} else if ( ( ((xc_temp - minx1x2) >= epsi_0) &&
 ((maxx1x2 - xc_temp) <= epsi_0) && ((yc_temp -
    miny1y2) >= epsi_0) &&
 ((maxy1y2 - yc_temp) <= epsi_0) ) ) {
 Prev_Cargo_exterior_interior_FLAG = 1.0;
} // End check if previous cargo position is on
   filament at its previous location

// Start check if current cargo position is off
   filament at its current location
// Find current filaments current max and min
Net_MINMAX_2(&minx1x2, &maxx1x2, &miny1y2, &maxy1y2
   , x1, x2, y1, y2);
Current_Cargo_exterior_FLAG = 0.0; // Set to False
   = 0.0, Only change it if the conditions are met
if ( ((xc != minx1x2) && (xc != maxx1x2) && (yc !=
   miny1y2) && (yc != maxy1y2)) ) {
 // Check horizontal case
 if (fabs(y2-y1) <= epsi_0) {
  if ( (xc < minx1x2) || (xc > maxx1x2) ) {
   Current_Cargo_exterior_FLAG = 1.0;
  }
 } else if (fabs(x2-x1) <= epsi_0) {
  // Check vertical case
  if ( (yc < miny1y2) || (yc > maxy1y2) ) {
   Current_Cargo_exterior_FLAG = 1.0;
  }
```

```
 } else if ( ((xc < minx1x2) && ( yc < miny1y2)) ||
     ((xc > maxx1x2) && (yc > maxy1y2)) ||
  ((xc < minx1x2) && (yc > maxy1y2)) || ((xc >
     maxx1x2) && (yc < miny1y2)) ) {
  Current_Cargo_exterior_FLAG = 1.0;
 }
} // End check if current cargo position is off
   filament at its current location

// Make sure endpoint case changes to perpendicular
    case if filament didn't change
if ( (Fil_ends_FLAG == 0.0) && (
   Current_Cargo_exterior_FLAG == 0.0) && (
   Prev_Cargo_exterior_interior_FLAG == 1.0) ) {
 // Change to perpendicular line case
 Fil_ends_FLAG = 1.0;
}

// Check if cargo is not within endpoint range of
   current filament
if ( (Current_Cargo_exterior_FLAG == 1.0) && (
   Prev_Cargo_exterior_interior_FLAG == 1.0) ) {

 // Only count the ends if cargo is not within
    buffer regions
 if ( ( (sqrt(pow((xc - xCellCent),2) + pow((yc -
    yCellCent),2))) > (inner + fil_buffer) ) &&
  ( (sqrt(pow((xc - xCellCent),2) + pow((yc -
     yCellCent),2))) < (outer - fil_buffer) ) ) {
  // Find which end cargo fell off filament (outer,
      inner, xCellCent, yCellCent)
  Net_FilOff_FB(&cOFF_front, &cOFF_back, filNet_M[m
     ][3], xc, yc, x1, x2, y1, y2, Cargo_prev_pos,
   Current_Fil_prev_pos, &Fil_ends_FLAG);
 }
```

```
        // Cargo has fallen off the network
        ON = 0; // Current cargo will not be on current
           filament starting next step
        OFF = 1; // Current cargo will be off current
           filament starting next step
        currentm = -1; // Not attached to any filament
        cOFF_end += 1.0; // Counter falling off from the
           ends of filament
        printf("Fell Off Ends, t = %d\n",tt);
       }
      }
     } // End check if cargo was still on network after
        filament modification

    } // End check if cargo in still inside cell membrane

    /* ############################################ */
    /* ################ CASE_A_6B ################ */
    /* ############################################ */
    // CASE_A_6B: Calculate networks average filament
       length

    for (int fav = 0; fav < numFils; fav++) {
     avg_fl += sqrt( pow((filEnds_M[fav][1] - filEnds_M[
        fav][0]),2) + pow((filEnds_M[fav][3] - filEnds_M[
        fav][2]),2) );
    }
    avg_fil_net[tt] = avg_fl/numFils;
    avg_fl = 0.0;

    /* ############################################ */
   } // End for loop movement of current cargo

   /* ############################################ */
   /* ################ CASE_A_7 ################ */
   /* ############################################ */
```

```
// CASE_A_7: Save current network/cargo data, record
   msd10/msd100, fraction time on network, Start FPTD
   calculations

/* ########################################### */
// Save network data, Free allocated memory space to
   prevent memory leakage

if (STOP == 1) {
 stepNum_2 = stepNum + 1; // New total time for cargo
} else {
 stepNum_2 = timeIntMax; // New total time for cargo
}

// Loop over all filaments
// Comment out if you want to do production run -> For
   filament shrink/growth VISUAL PLOTTING
/* ----------------------------------------- */
for (int cc = 0; cc < numFils; cc++) {

 // Re-declare character string in loop so it doesn't
    append next filament number
 char st_begFL[1024] = "NETWORK_SG_DATA";
 char st_endFL[1024];

 // Open FILE to write data too
 FILE *outdata;
 sprintf(st_endFL,"_FPau%.2fFCatP%.4fFResP%.4f_NetNUM%
    d_Filnum%d.txt",(f_MT_PAUSE),(sg_prob_M_Cat),(
    sg_prob_M_Res),(currentNet+1),(cc+1)); // Append
    filament number for data
 outdata = fopen(strcat(st_begFL,st_endFL),"w");

 // Loop over all time for current filament
 for (int ttt = 0; ttt < (stepNum_2); ttt++) {
```

```
  // Store current filaments x1,x2,y1,y2,p at current
     time
  fprintf(outdata,"%lf\t%lf\t%lf\t%lf\t%lf\n",
     filEnds_M2[ttt][cc][0],filEnds_M2[ttt][cc][1],
     filEnds_M2[ttt][cc][2],filEnds_M2[ttt][cc][3],
     filNet_M2[ttt][cc][3]);
 }
 // Close FILE of current filament
 fclose(outdata);
} // End for loop over all filaments

// De-allocate memory of filament arrays
for (int ii = 0; ii < timeIntMax; ii++) {
 for (int jj = 0; jj < numFils; jj++) {
  free(filNet_M2[ii][jj]);
  free(filEnds_M2[ii][jj]);
 }
}
free(filNet_M2);
free(filEnds_M2);
/* ------------------------------------------ */

/* ########################################## */
// Save cargo data, Free allocated memory space to
   prevent memory leakage

// Comment out if you want to do production run -> For
   cargo VISUAL PLOTTING
/* ------------------------------------------ */
// Re-declare character string in loop so it doesn't
   append next cargo number
char st_begCO[1024] = "NETWORK_SG_DATA";
char st_endCO[1024];

// Open FILE to write data too
FILE *outdata;
```

```
sprintf(st_endCO,"_FPau%.2fFCatP%.4fFResP%.4f_NetNUM%
   d_Cargonum%d.txt",(f_MT_PAUSE),(sg_prob_M_Cat),(
   sg_prob_M_Res),(currentNet+1),(currentCarg+1)); //
   Append cargo number for data
outdata = fopen(strcat(st_begCO,st_endCO),"w");
// Loop over all time for current cargo
for (int ccc = 0; ccc < (stepNum_2); ccc++) {
 // Store current cargo x,y position current time
 fprintf(outdata,"%lf\t%lf\n",Cargo_pos[ccc][0],
    Cargo_pos[ccc][1]);
} // End for loop over all cargo time
// Close FILE of current cargo
fclose(outdata);

// De-allocate memory of Cargo arrays
for (int ii = 0; ii < timeIntMax; ii++) {
 free(Cargo_pos[ii]);
}
free(Cargo_pos);
/* ---------------------------------------- */

/* ######################################### */
// Record msd10 and msd100 for cargo in current network
   , Start FPTD calculations

// Record total msd of current network
msd10[currentNet] += msdCurrent10;
msd100[currentNet] += msdCurrent100;

// Record cargo information -> (cOFF_front, cOFF_back,
   ckOFF, ckON, ckON_tot, cSwitch, cSwitch_tot, tOn,
   tOff, t)
CargoInfo[currentCargo][0] = cOFF_front; // Total count
    falling off front of filament
CargoInfo[currentCargo][1] = cOFF_back; // Total count
   falling off back of filament
```

```
CargoInfo[currentCargo][2] = ckOFF; // Total count kOff
    executed -> Cargo fell off filament (Not the end
   points meaning walking off)
CargoInfo[currentCargo][3] = ckON; // Total count kOn
   executed -> Cargo attached to filament
CargoInfo[currentCargo][4] = ckON_tot; // Total count
   of total chances of attaching to filaments
CargoInfo[currentCargo][5] = cSwitch; // Total count
   switching to different filament
CargoInfo[currentCargo][6] = cSwitch_tot; // Total
   count of total chances of switching to different
   filaments
CargoInfo[currentCargo][7] = tOn; // Cargo total time
   ballistic motion on filament (seconds)
CargoInfo[currentCargo][8] = tOff; // Cargo total time
   anomalous diffusion off filament (seconds)
CargoInfo[currentCargo][9] = t; // Total physical
   simulation time (seconds)
CargoInfo[currentCargo][10] = ckOFF_tot; // Total count
    of total chances of detaching from filaments
CargoInfo[currentCargo][11] = cOFF_end; // Counter
   falling off from the ends of filament

// Record fraction of time on data
fracTimeOn[currentCargo] = tOn / t;
currentCargo += 1; // Move to next cargo

// Only Record FPTD if cargo reached cell edge
if (STOP == 1) {
 // A cargo escaped at timestep stepNum = tt -> Add one
     to the time escape happened
 FPTD[stepNum] = FPTD[stepNum] + 1;
 // Time cargo left cell -> FPT
 cargoFPTs[currentCarg] = stepNum_time;

 // Counter for 10 and 100 seconds
```

```
  if (t <= 10.0) {
   count10++;
  } else if (t <= 100.0) {
   count100++;
  }
 }

} // End for loop over all cargos for simulation

/* ########################################### */
/* ################# CASE_A_8 ################ */
/* ########################################### */
// CASE_A_8: Calculate msd10/msd100 for all cargo in
   network, Save cargoFPTs of network, Start MFPT
   calculations for network

// Msd for all cargos at 10s and 100s
msd10[currentNet] = msd10[currentNet] / numCargs;
msd100[currentNet] = msd100[currentNet] / numCargs;

// Data for production run
/* ----------------------------------------- */
if (production_run == 1) {
 // Output file -1 -> cargoFPT file -> Each column is a
    new network
 char sBeg[1024] = "Cargo_FPT";
 char sBeg2[1024];
 sprintf(sBeg2,"_FPau%.2fFCatP%.4fFResP%.4fNetNUM%d",(
    f_MT_PAUSE),(sg_prob_M_Cat),(sg_prob_M_Res),(
    currentNet+1));
 strcat(sBeg,sBeg2); // append sBeg2 to char sBeg

 FILE *outcFPT;
 outcFPT = fopen(strcat(sBeg,sEnd1),"w");
 for (int i = 0; i < numCargs; i++) {
  fprintf(outcFPT,"%lf\n",cargoFPTs[i]);
```

```
 }
 fclose(outcFPT);

 // Output file 0 -> cargoFPT file -> Each column is a
    new network
 char sBegFAL[1024] = "Fil_AVGL";
 char sBeg2FAL[1024];
 sprintf(sBeg2FAL,"_FPau%.2fFCatP%.4fFResP%.4fNetNUM%d
    ",(f_MT_PAUSE),(sg_prob_M_Cat),(sg_prob_M_Res),(
    currentNet+1));
 strcat(sBegFAL,sBeg2FAL); // append sBeg2 to char sBeg

 FILE *outcFAL;
 outcFAL = fopen(strcat(sBegFAL,sEnd1),"w");
 for (int i = 0; i < timeIntMax; i++) {
  fprintf(outcFAL,"%lf\n",avg_fil_net[i]);
 }
 fclose(outcFAL);
}
/* ---------------------------------------- */

// Count number of cargos that left cell with respective
    time -> 10 and 100 seconds
fluxOut10[currentNet] = count10;
fluxOut100[currentNet] = count100;

// Calculate MFPT for all cargos on this network
fptSum = 0.0;
for(int i = 0; i < numCargs; i++){
 fptSum += cargoFPTs[i];
}

// Calculate FPTs average, varaince and standard
   deviation of current network
fptVar = 0.0;
```

```
 cargoMFPTs[currentNet] = fptSum / numCargs; // MFPT for
    this network -> (SUM (fpt) )/ (number of cargos)
 for(int i = 0; i < numCargs; i++){
  fptVar += pow((cargoFPTs[i] - cargoMFPTs[currentNet])
     ,2) / numCargs;
 }
 cargoFPTstdev[currentNet] = sqrt(fptVar); // Standard
    deviation of FPTs for this network
 //printf("flux out 10: %d flux out 100: %d\n",count10,
    count100);

} // End for loop over all networks

/* ########################################### */
/* ################# CASE_B_1 ################ */
/* ########################################### */
// CASE_B_1: Do MSD analysis

// NOTE: DATA OUTPUT IS WRT VARIATION IN FILAMENT LENGTHS
    AND NUMBER OF FILAMENTS.
//       ASSUMES CARGOS ARE INDEPENDENT THUS FOR EACH
   NETWORK WITH SOME FIL LENGTH AND NUMBER OF FILS, WITH
//       SOME NUM OF CARGOS DIFFUSING/WALKING, MSD IS
   COMPUTED FOR THAT RESPECTIVE FIL LENGTH VALUE AND
   NUMBER OF FILAMENTS

// Data for production run
/* ---------------------------------------- */
if (production_run == 1) {
 // Add pause and catastrophy freq to all text file name
 char sBegA[1024];
 sprintf(sBegA,"_FPau%.2fFCatP%.4fFResP%.4f",(f_MT_PAUSE)
    ,(sg_prob_M_Cat),(sg_prob_M_Res));

 // MSD
 msdSum10 = 0.0, msdSum100 = 0.0;
```

```
var10 = 0.0, var100 = 0.0;
for (int i = 0; i < numNets; i++) {
 msdSum10 += msd10[i];
 msdSum100 += msd100[i];
}

// Calculate average MSD, variation and standard
   deviation
av10 = msdSum10 / numNets, av100 = msdSum100/ numNets;
for (int i = 0; i < numNets; i++) {
 var10 += pow((msd10[i] - av10), 2) / numNets;
 var100 += pow((msd100[i] - av100), 2) / numNets;
}
stdev10 = sqrt(var10); // If only 1 network, std and var
    will be zero
stdev100 = sqrt(var100);

// Output file 1 and 2 -> MSD STANDARD DEVIATION files
   -> FILES/DATA NEVER USED IN ANALYSIS FROM BRYAN
   MAELFEYT
FILE *outp10;
FILE *outp100;

char sBeg10[1024] = "msdSTDEV10";
char sBeg100[1024] = "msdSTDEV100";
strcat(sBeg10,sBegA); // Append sBegA to char sBeg10
strcat(sBeg100,sBegA); // Append sBegA to char sBeg100

outp10 = fopen(strcat(sBeg10,sEnd1),"w");
outp100 = fopen(strcat(sBeg100,sEnd1),"w");

fprintf(outp10,"%lf\n",stdev10);
fprintf(outp100,"%lf\n",stdev100);

fclose(outp10);
fclose(outp100);
```

```
// Output file 3 -> MSD file -> ONLY FILE/DATA USED BY
   BRYAN MAELFEYT IN ANALYSIS
FILE *outp;
char sBegMSD[1024] = "MSD";
strcat(sBegMSD,sBegA); // Append sBegA to char sBegMSD
outp = fopen(strcat(sBegMSD,sEnd1),"w");
for (int i = 0; i < timeIntMax; i++) {
 fprintf(outp,"%lf\t%lf\t%lf\n",msdArray[i][0],msdArray[
    i][1],msdArray[i][2]);
}
fclose(outp);

/* ########################################### */
/* ################ CASE_B_2 ################ */
/* ########################################### */
// CASE_B_2: Do MFPT analysis

// Calculate overall MFPT
mfptSum = 0.0;
for (int i = 0; i < numNets; i++) {
 mfptSum += cargoMFPTs[i];
}

// Calculate MFPT average, variance and standard
   deviation over all networks
mfptVar = 0.0;
totMFPT = mfptSum / numNets;
for (int i = 0; i < numNets; i++) {
 mfptVar += pow((cargoMFPTs[i] - totMFPT),2) / numNets;
}
totStdev = sqrt(mfptVar); // If only 1 network, std and
   var will be zero

// Calculate FPTs average standard deviation
stdevSum = 0.0;
```

```
for (int i = 0; i < numNets; i++) {
 stdevSum += cargoFPTstdev[i];
}
avgStdev = stdevSum / numNets;

// Calculate overall Flux
fluxSum10 = 0.0, fluxSum100 = 0.0;
fluxvar10 = 0.0, fluxvar100 = 0.0;
for (int i = 0; i < numNets; i++) {
 fluxSum10 += fluxOut10[i];
 fluxSum100 += fluxOut100[i];
}

// Calculate Flux average, variation and standard
   deviation
fluxav10 = (fluxSum10 / numNets), fluxav100 = (
   fluxSum100 / numNets);
for (int i = 0; i < numNets; i++) {
 fluxvar10 += pow((fluxOut10[i] - fluxav10),2) / numNets
    ;
 fluxvar100 += pow((fluxOut100[i] - fluxav100),2) /
    numNets;
}
fluxstdev10 = sqrt(fluxvar10); // If only 1 network, std
    and var will be zero
fluxstdev100 = sqrt(fluxvar100);

// Output file 4 -> MFPT, STD MFPT and AVERAGE STD MFPT
   file
FILE *outpMFPT;
char sBegMFPT[1024] = "infoMFPT";
strcat(sBegMFPT,sBegA); // Append sBegA to char sBegMFPT
outpMFPT = fopen(strcat(sBegMFPT,sEnd1),"w");
fprintf(outpMFPT,"Overall MFPT:\t%lf\nMFPT standard
   deviation:\t%lf\nAverage standard deviation:\t%lf\n",
   totMFPT,totStdev,avgStdev);
```

```
fclose(outpMFPT);

// Output file 5 and 6 -> Flux AVERAGE and STANDARD
   DEVIATION files
FILE *foutp10;
FILE *foutp100;
char sBegfl10[1024] = "fluxout10";
char sBegfl100[1024] = "fluxout100";
strcat(sBegfl10,sBegA); // Append sBegA to char sBegfl10
strcat(sBegfl100,sBegA); // Append sBegA to char
   sBegfl100
foutp10 = fopen(strcat(sBegfl10,sEnd1),"w");
foutp100 = fopen(strcat(sBegfl100,sEnd1),"w");
fprintf(foutp10,"%lf\t%lf\n",fluxav10,fluxstdev10);
fprintf(foutp100,"%lf\t%lf\n",fluxav100,fluxstdev100);
fclose(foutp10);
fclose(foutp100);

// Output file 7 -> FIRST PASSAGE TIME DISTRUTION file
FILE *FPoutp;
char sBegFPTD[1024] = "FPTD";
strcat(sBegFPTD,sBegA); // Append sBegA to char sBegFPTD
FPoutp = fopen(strcat(sBegFPTD,sEnd1),"w");
for (int i = 0; i < timeIntMax; i++) {
  fprintf(FPoutp,"%lf\n",FPTD[i]);
}
fclose(FPoutp);

// Output file 8 -> FRACTION OF TIME ON FILAMENT file ->
    FILES/DATA NEVER USED IN ANALYSIS FROM BRYAN
   MAELFEYT
FILE *outpFracs;
char sBegFracs[1024] = "fracTimeOnMSD";
strcat(sBegFracs,sBegA); // Append sBegA to char
   sBegFracs
outpFracs = fopen(strcat(sBegFracs,sEnd1),"w");
```

```
      for (int i = 0; i < numCargs*numNets; i++) {
       fprintf(outpFracs,"%lf\n",fracTimeOn[i]);
      }
      fclose(outpFracs);

      // Output file 9 -> CARGO INFORMATION -> (cOFF_front,
         cOFF_back, ckOFF, ckON, ckON_tot, cSwitch,
         cSwitch_tot, tOn, tOff, t, ckOFF_tot, cOFF_end)
      FILE *outpCargoInf;
      char sBegCargoInf[1024] = "CargoInfo";
      strcat(sBegCargoInf,sBegA); // Append sBegA to char
         sBegCargoInf
      outpCargoInf = fopen(strcat(sBegCargoInf,sEnd1),"w");
      for (int i = 0; i < numCargs*numNets; i++) {
       fprintf(outpCargoInf,"%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf
          \t%lf\t%lf\t%lf\t%lf\t%lf\n",CargoInfo[i][0],
          CargoInfo[i][1],CargoInfo[i][2],
        CargoInfo[i][3],CargoInfo[i][4],CargoInfo[i][5],
           CargoInfo[i][6],CargoInfo[i][7],CargoInfo[i][8],
           CargoInfo[i][9],CargoInfo[i][10],
        CargoInfo[i][11]);
      }
      fclose(outpCargoInf);
     }
     /* ---------------------------------------- */

     /* ######################################## */
    } // End for loop over all number of filaments

   } // End for loop over all filament lengths

  } // End for loop over all cargo switching probability

} // End for loop over all filament rescue probability

// System total run time calculation
```

```
 double t2 = clock();
 double total_time = (t2-t1)/CLOCKS_PER_SEC;

 // Save system total run time
 FILE *outpTIME;
 char sBegTIME[1024] = "Total_RunTIME.txt";
 outpTIME = fopen(sBegTIME,"w");
 fprintf(outpTIME,"%lf",total_time);
 fclose(outpTIME);

 // Exit
 return 0;

} // End main
```

B.3.2 Net_Setup.c

This program is used in the main code to setup the filament network topology.

```
// Net_Setup.c: Imtiaz Ali
// Description: Function to make network, RNG/Negative/
   Positive Polarity and Non-Radial/Radial Distribution
// Compiler: GCC 8.1
// Last modified: 02/21/2020
// Input: or = outer cell radius, ir = inner cell radius, fl =
    filament length, xcent = cell x center, ycent = cell y
   center

#include "Net_Setup.h"
#define M_PI 3.14159265358979323846

void Net_Setup(size_t j, size_t i, float Net_fil[j][i], float
   Ends_fil[j][i], float or,
  float ir, float fl, float xcent, float ycent, int numFils,
     int fil_radial, int fil_pol, float fil_buffer) {
```

```
// initialize local variables
float x_1, x_2, y_1, y_2, df_1;
float r_1, r_2, p_1, theta_1, alpha_1;
float d_change = fil_buffer + 0.1; // (microns) Closest
   distance to inner or outer regions (buffer)

// Distribution of filament
if (fil_radial == 1) {
  // Uniform  radial starting position
  r_1 = ir + d_change;
  // Uniform angular starting position
  // theta_1 = (M_PI/180)*30*j;
  theta_1 = (M_PI/180)*(360/numFils)*j;
  // Alpha
  alpha_1 = 0;
} else {
  // Random radial starting position
  r_1 = or - (or - ir) * (float)rand()/RAND_MAX;
  // Random angular starting position
  theta_1 = (2*M_PI) * (float)rand()/RAND_MAX;
  // Alpha -> between -pi/2 and +pi/2
  alpha_1 = -(M_PI) * (float)rand()/RAND_MAX + (M_PI/2);
}

// Set filament polarity -> positive is "out" negative is "
   in"
if (fil_pol == -1) {
  p_1 = -1; // Negative polarity
} else if (fil_pol == 1) {
  p_1 = 1; // Positive polarity
} else {
  p_1 = (-2) * (float)rand()/RAND_MAX + 1; // Random
     polarity
}
p_1 = p_1 / fabs(p_1); // Make into 1 or -1 -> p is +1 or -1
```

```
// x and y values of filament endpoints -> start from center
    -> radius for next point is filament length
// theta = initial angle, alpha = difference from initial
   angle
x_1 = xcent + r_1 * cos(theta_1);
y_1 = ycent + r_1 * sin(theta_1);
x_2 = x_1 + fl * cos(theta_1 + alpha_1);
y_2 = y_1 + fl * sin(theta_1 + alpha_1);
r_2 = sqrt(pow((x_2-xcent),2)+pow((y_2-ycent),2)); // "outer
   " end of filament

// make sure filament ends are within desired region
// shift filament out
if(r_1 < (ir + d_change)){
  df_1 = (ir + d_change) - r_1; // add the difference of
     being within inner radius
  r_1 = r_1 + df_1;
  r_2 = r_2 + df_1;
}
// shift filament in
if(r_2 > (or - d_change)){
  df_1 = r_2 - (or - d_change); // subtract the difference
     of being outside of outer radius
  r_1 = r_1 - df_1;
  r_2 = r_2 - df_1;
}

// adjusted x and y values of filament endpoints, x and y
   values of filament endpoints
x_1 = xcent + r_1 * cos(theta_1);
y_1 = ycent + r_1 * sin(theta_1);
x_2 = x_1 + fl * cos(theta_1 + alpha_1);
y_2 = y_1 + fl * sin(theta_1 + alpha_1);
// "outer" end of filament -> NOT SURE WHY BRYAN DIDN'T SAVE
    r2!!??
r_2 = sqrt(pow((x_2 - xcent),2) + pow((y_2 - ycent),2));
```

```
  // set return array values
  Net_fil[j][0] = r_1; Net_fil[j][1] = theta_1; Net_fil[j][2]
     = alpha_1; Net_fil[j][3] = p_1;
  Ends_fil[j][0] = x_1; Ends_fil[j][1] = x_2; Ends_fil[j][2] =
      y_1; Ends_fil[j][3] = y_2;

}
```

B.3.3 Net_Distances_MINMAX.c

This program is used in the main to calculate cargo distance from filament.

```
// Net_Distances.c: Imtiaz Ali
// Description: Function to make perpendicular distance,
   distance from both ends of filament and finding max and min
    grid ranges
// compiler: GCC 8.1
// last modified: 01/08/2021
// Input: d_p = perp dist of cargo to fil segment, d_1 = dist
   to fil seg start, d_2 = dist to fil seg end, x_carg = cargo
    x pos, y_carg = cargo y pos
//        Ends_fil = fil segment start and end (x,y) pos
//        minx_1 = fil seg min x pos, maxx_2 = fil seg max x
   pos, miny_1 = fil seg min y pos, maxy_2 = fil seg max y pos
//        x_1 = current fil seg start x pos, x_2 = current fil
    seg end x pos, y_1 = current fil seg start y pos, y_2 =
   current fil seg end y pos,

#include <stdio.h> // NULL
#include "Net_Distances_MINMAX.h"
#define M_PI 3.14159265358979323846

void Net_Distances(float* d_p, float* d_1, float* d_2, float
   x_carg, float y_carg, size_t row, size_t colm, float
   Ends_fil[row][colm]) {
```

```
 // Perpendicular distance from a point to a point on a line
 *d_p = fabs((Ends_fil[row][3] - Ends_fil[row][2]) * x_carg -
    (Ends_fil[row][1] - Ends_fil[row][0]) * y_carg
    + Ends_fil[row][1] * Ends_fil[row][2] - Ends_fil[row][3] *
        Ends_fil[row][0]) /
    sqrt(pow((Ends_fil[row][3] - Ends_fil[row][2]),2) + pow((
       Ends_fil[row][1] - Ends_fil[row][0]),2));
 // Current cargo distance from current filaments inner
    endpoint -> d1
 *d_1 = sqrt(pow((x_carg - Ends_fil[row][0]),2) + pow((y_carg
    - Ends_fil[row][2]),2));
 // Current cargo distance from current filaments outer
    endpoint -> d2
 *d_2 = sqrt(pow((x_carg - Ends_fil[row][1]),2) + pow((y_carg
    - Ends_fil[row][3]),2));
}

void Net_MINMAX_1(float* minx_1, float* maxx_2, float* miny_1,
    float* maxy_2, size_t row, size_t colm, float Ends_fil[row
   ][colm]) {
 // // Old version, don't use (causes numerical round off
    errors)
  //  // Current filament segments min/max relative (x,y)
     positions
 // *minx_1 = 0.5 * fabs(Ends_fil[row][0] + Ends_fil[row][1])
    - 0.5 * fabs(Ends_fil[row][0] - Ends_fil[row][1]);
 // *maxx_2 = 0.5 * fabs(Ends_fil[row][0] + Ends_fil[row][1])
    + 0.5 * fabs(Ends_fil[row][0] - Ends_fil[row][1]);
 // *miny_1 = 0.5 * fabs(Ends_fil[row][2] + Ends_fil[row][3])
    - 0.5 * fabs(Ends_fil[row][2] - Ends_fil[row][3]);
 // *maxy_2 = 0.5 * fabs(Ends_fil[row][2] + Ends_fil[row][3])
    + 0.5 * fabs(Ends_fil[row][2] - Ends_fil[row][3]);

  // Current filament segments min/max relative (x,y)
     positions
  if (Ends_fil[row][0] == Ends_fil[row][1]) {
```

```
    *minx_1 = Ends_fil[row][0];
    *maxx_2 = Ends_fil[row][0];
  } else if (Ends_fil[row][0] < Ends_fil[row][1]) {
    *minx_1 = Ends_fil[row][0];
    *maxx_2 = Ends_fil[row][1];
  } else {
    *minx_1 = Ends_fil[row][1];
    *maxx_2 = Ends_fil[row][0];
  }
  if (Ends_fil[row][2] == Ends_fil[row][3]) {
    *miny_1 = Ends_fil[row][2];
    *maxy_2 = Ends_fil[row][2];
  } else if (Ends_fil[row][2] < Ends_fil[row][3]) {
    *miny_1 = Ends_fil[row][2];
    *maxy_2 = Ends_fil[row][3];
  } else {
    *miny_1 = Ends_fil[row][3];
    *maxy_2 = Ends_fil[row][2];
  }
}

void Net_MINMAX_2(float* minx_1, float* maxx_2, float* miny_1,
    float* maxy_2, float x_1, float x_2, float y_1, float y_2)
    {
  // // Old version, don't use (causes numerical round off
     errors)
 // *minx_1 = 0.5 * fabs(x_1 + x_2) - 0.5 * fabs(x_1 - x_2);
 // *maxx_2 = 0.5 * fabs(x_1 + x_2) + 0.5 * fabs(x_1 - x_2);
 // *miny_1 = 0.5 * fabs(y_1 + y_2) - 0.5 * fabs(y_1 - y_2);
 // *maxy_2 = 0.5 * fabs(y_1 + y_2) + 0.5 * fabs(y_1 - y_2);

  // Current filament segments min/max relative (x,y)
     positions
  if (x_1 == x_2) {
    *minx_1 = x_1;
    *maxx_2 = x_1;
```

```
  } else if (x_1 < x_2) {
    *minx_1 = x_1;
    *maxx_2 = x_2;
  } else {
    *minx_1 = x_2;
    *maxx_2 = x_1;
  }
  if (y_1 == y_2) {
    *miny_1 = y_1;
    *maxy_2 = y_1;
  } else if (y_1 < y_2) {
    *miny_1 = y_1;
    *maxy_2 = y_2;
  } else {
    *miny_1 = y_2;
    *maxy_2 = y_1;
  }
}

void Net_FilOff_FB(float* coff_f, float* coff_b, float p_1,
   float x_c, float y_c,
 float x_1, float x_2, float y_1, float y_2, float
    Cargo_prev_pos[], float Current_Fil_prev_pos[], float*
    Fil_ends_FLAG) {

 // Initialize variables to hold (+END) and (-END) of filament
     positions
 float x_beg, y_beg, x_end, y_end, x_beg_prev, y_beg_prev,
    x_end_prev, y_end_prev;

  // Assign the (-END) = Beginning of filament and (+END) =
     Ending of filament => (INDEPENDENT OF MOTOR TYPE)
  /* ------------------------------------------*/
  if (p_1 < 0) {
    // Negative polarity
    // Current filament
```

```
    x_beg = x_2; // (-END)
    y_beg = y_2; // (-END)
    x_end = x_1; // (+END)
    y_end = y_1; // (+END)

    // Previous filament
    x_beg_prev = Current_Fil_prev_pos[1]; // (-END)
    y_beg_prev = Current_Fil_prev_pos[3]; // (-END)
    x_end_prev = Current_Fil_prev_pos[0]; // (+END)
    y_end_prev = Current_Fil_prev_pos[2]; // (+END)
  } else if (p_1 > 0) {
    // Positive polarity
    // Current filament
    x_beg = x_1; // (-END)
    y_beg = y_1; // (-END)
    x_end = x_2; // (+END)
    y_end = y_2; // (+END)

    // Previous filament
    x_beg_prev = Current_Fil_prev_pos[0]; // (-END)
    y_beg_prev = Current_Fil_prev_pos[2]; // (-END)
    x_end_prev = Current_Fil_prev_pos[1]; // (+END)
    y_end_prev = Current_Fil_prev_pos[3]; // (+END)
  }
  /* -----------------------------------------*/

  // Check if working with endpoints
  if (*Fil_ends_FLAG == 0.0) {
    // Distance method needs to me used if endpoint conditions
        met
    // Previous filament to current cargo (One of these will
       be zero)
    float d_B2A = sqrt(pow((x_beg_prev - x_c),2) + pow((
       y_beg_prev - y_c),2));
    float d_E2A = sqrt(pow((x_end_prev - x_c),2) + pow((
       y_end_prev - y_c),2));
```

```
  // Current filament to current cargo
  float d_B2B = sqrt(pow((x_beg - x_c),2) + pow((y_beg - y_c
     ),2));
  float d_E2B = sqrt(pow((x_end - x_c),2) + pow((y_end - y_c
     ),2));

  if ( (((d_E2B > d_E2A) && (d_E2A == 0.0)) && ((d_E2B <
     d_B2A) && (d_E2B < d_B2B))) ||
    (((d_E2B < d_B2A) && (d_E2B < d_B2B))) ) {
    // Fell off at filament FRONT
    *coff_f += 1.0; // (+END)
    *Fil_ends_FLAG = -1.0;
  } else if ( (((d_B2B > d_B2A) && (d_B2A == 0.0)) && ((
     d_B2B < d_E2A) && (d_B2B < d_E2B))) ||
    ((d_B2B < d_E2A) && (d_B2B < d_E2B)) ) {
    // Fell off at filament BACK
    *coff_b += 1.0; // (-END)
    *Fil_ends_FLAG = -1.0;
  }
}

// Check if working within endpoints
if (*Fil_ends_FLAG == 1.0) {
  // Vector conditions will work if perpendicular line
     condition met

  // Initialize variables to hold dot-product computations
  float d_c[2], d_B1[2], d_B2A[2], d_B2B[2], d_E1[2], d_E2A
     [2], d_E2B[2]; // Difference vector array
  float CP_Norm[6]; // Ratio of normalized dot product -> [
     d_c*d_B1, d_c*d_B2A, d_c*d_B2B, d_c*d_E1, d_c*d_E2A,
     d_c*d_E2B]
  float Diff_Mag[7]; // Magnitude of each difference vector
     -> [d_c, d_B1, d_B2A, d_B2B, d_E1, d_E2A, d_E2B]
```

```
float zero_flag[4] = {0.0,0.0,0.0,0.0}; // Flag for
   division by zero (4 POSSIBLE CASES) -> [d_B1, d_B2A,
   d_E1, d_E2A], FALSE = 0, TRUE = 1

// Assign difference vectors
/* ------------------------------------------*/
// Cargo-to-Cargo (Current_Cargo - Previous_Cargo)
d_c[0] = x_c - Cargo_prev_pos[0]; // Cargo (xc_2 - xc_1)
d_c[1] = y_c - Cargo_prev_pos[1]; // Cargo (yc_2 - yc_1)
// Cargo-to-Filamnt (Previous_Fil_Beg/End - Previous_Cargo
   )
d_B1[0] = x_beg_prev - Cargo_prev_pos[0]; // Filament (
   xbeg_prev - xc_1)
d_B1[1] = y_beg_prev - Cargo_prev_pos[1]; // Filament (
   ybeg_prev - yc_1)
d_E1[0] = x_end_prev - Cargo_prev_pos[0]; // Filament (
   xend_prev - xc_1)
d_E1[1] = y_end_prev - Cargo_prev_pos[1]; // Filament (
   yend_prev - yc_1)
// Cargo-to-Filamnt (Current_Fil_Beg/End - Previous_Cargo)
d_B2A[0] = x_beg - Cargo_prev_pos[0]; // Filament (xbeg -
   xc_1)
d_B2A[1] = y_beg - Cargo_prev_pos[1]; // Filament (ybeg -
   yc_1)
d_E2A[0] = x_end - Cargo_prev_pos[0]; // Filament (xend -
   xc_1)
d_E2A[1] = y_end - Cargo_prev_pos[1]; // Filament (yend -
   yc_1)
// Cargo-to-Filamnt (Current_Fil_Beg/End - Current_Cargo)
d_B2B[0] = x_beg - x_c; // Filament (xbeg - xc_2)
d_B2B[1] = y_beg - y_c; // Filament (ybeg - yc_2)
d_E2B[0] = x_end - x_c; // Filament (xend - xc_2)
d_E2B[1] = y_end - y_c; // Filament (yend - yc_2)
/* ------------------------------------------*/

// Compute difference vector magnitude
```

```
/* ------------------------------------------*/
Diff_Mag[0] = sqrt(pow(d_c[0],2) + pow(d_c[1],2)); // d_c
Diff_Mag[1] = sqrt(pow(d_B1[0],2) + pow(d_B1[1],2)); //
   d_B1
Diff_Mag[2] = sqrt(pow(d_B2A[0],2) + pow(d_B2A[1],2)); //
   d_B2A
Diff_Mag[3] = sqrt(pow(d_B2B[0],2) + pow(d_B2B[1],2)); //
   d_B2B
Diff_Mag[4] = sqrt(pow(d_E1[0],2) + pow(d_E1[1],2)); //
   d_E1
Diff_Mag[5] = sqrt(pow(d_E2A[0],2) + pow(d_E2A[1],2)); //
   d_E2A
Diff_Mag[6] = sqrt(pow(d_E2B[0],2) + pow(d_E2B[1],2)); //
   d_E2B
/* ------------------------------------------*/

// Set flag for division by zero
/* ------------------------------------------*/
if (Diff_Mag[1] == 0) {
  zero_flag[0] = 1.0; // d_B1
}
if (Diff_Mag[2] == 0) {
  zero_flag[1] = 1.0; // d_B2A
}
if (Diff_Mag[4] == 0) {
  zero_flag[2] = 1.0; // d_E1
}
if (Diff_Mag[5] == 0) {
  zero_flag[3] = 1.0; // d_E2A
}
/* ------------------------------------------*/

// Condition (Dynein is the opposite of the Kinesin/Myosin
    condition)
// Dynamic filament case
```

```
// Fall off BACK (Kinesin/Myosin) => (d_c*d_E1 == d_c*
   d_E2A == d_c*d_E2B = 1) && (d_c*d_B2A == d_c*d_B2B = 1)
    && (d_c*d_B1 = -1)
// Fall off FRONT (Kinesin/Myosin) => (d_c*d_B1 == d_c*
   d_B2A == d_c*d_B2B = -1) && (d_c*d_E1 == d_c*d_E2A = 1)
    && (d_c*d_E2B = -1)
// Fall off BACK (Dynein) => (d_c*d_E1 == d_c*d_E2A == d_c
   *d_E2B = -1) && (d_c*d_B2A == d_c*d_B2B = -1) && (d_c*
   d_B1 = 1)
// Fall off FRONT (Dynein) => (d_c*d_B1 == d_c*d_B2A ==
   d_c*d_B2B = 1) && (d_c*d_E1 == d_c*d_E2A = -1) && (d_c*
   d_E2B = 1)
// Static or no modified filament case (We don't need to
   worry about cargo type -> d_c takes care of it)
// Fall off BACK (Kinesin/Myosin/Dynein) => (d_c*d_E1 ==
   d_c*d_E2A == d_c*d_E2B = -1) && (d_c*d_B1 == d_c*d_B2A
   = 1) && (d_c*d_B2B = -1)
// Fall off FRONT (Kinesin/Myosin/Dynein) => (d_c*d_B1 ==
   d_c*d_B2A == d_c*d_B2B = -1) && (d_c*d_E1 == d_c*d_E2A
   = 1) && (d_c*d_E2B = -1)

// Compute Cross-Product Normalized Ration => (d_vec DOT
   a_vec)/(norm(d_vec) * norm(a_vec)) = cos(theta_{d_vec,
   a_vec})
/* -----------------------------------------*/
if (zero_flag[0] == 0.0) {
  CP_Norm[0] = (d_c[0]*d_B1[0] + d_c[1]*d_B1[1]) / (
     Diff_Mag[0]*Diff_Mag[1]); // d_c*d_B1 -> (
     Previous_Fil_Beg - Previous_Cargo)
} else {
  CP_Norm[0] = 0.0;
}
if (zero_flag[1] == 0.0) {
  CP_Norm[1] = (d_c[0]*d_B2A[0] + d_c[1]*d_B2A[1]) / (
     Diff_Mag[0]*Diff_Mag[2]); // d_c*d_B2A -> (
     Current_Fil_Beg - Previous_Cargo)
```

```
} else {
  CP_Norm[1] = 0.0;
}
CP_Norm[2] = (d_c[0]*d_B2B[0] + d_c[1]*d_B2B[1]) / (
   Diff_Mag[0]*Diff_Mag[3]); // d_c*d_B2B -> (
   Current_Fil_Beg - Current_Cargo)

if (zero_flag[2] == 0.0) {
  CP_Norm[3] = (d_c[0]*d_E1[0] + d_c[1]*d_E1[1]) / (
     Diff_Mag[0]*Diff_Mag[4]); // d_c*d_E1 -> (
     Previous_Fil_End - Previous_Cargo)
} else {
  CP_Norm[3] = 0.0;
}
if (zero_flag[3] == 0.0) {
  CP_Norm[4] = (d_c[0]*d_E2A[0] + d_c[1]*d_E2A[1]) / (
     Diff_Mag[0]*Diff_Mag[5]); // d_c*d_E2A -> (
     Current_Fil_End - Previous_Cargo)
} else {
  CP_Norm[4] = 0.0;
}
CP_Norm[5] = (d_c[0]*d_E2B[0] + d_c[1]*d_E2B[1]) / (
   Diff_Mag[0]*Diff_Mag[6]); // d_c*d_E2B -> (
   Current_Fil_End - Current_Cargo)
/* ------------------------------------------*/

// Check Front than Back
/* ------------------------------------------*/
if ( (((CP_Norm[0] < 0) && (CP_Norm[1] < 0) && (CP_Norm[2]
    < 0)) && ((CP_Norm[3] >= 0) && (CP_Norm[4] >= 0)) && (
   CP_Norm[5] < 0)) ||
(((CP_Norm[0] < 0) && (CP_Norm[1] < 0) && (CP_Norm[2] < 0)
   ) && ((CP_Norm[3] >= 0) && (CP_Norm[4] > 0)) && (
   CP_Norm[5] < 0)) ||
```

```
    (((CP_Norm[0] > 0) && (CP_Norm[1] > 0) && (CP_Norm[2] > 0)
       ) && ((CP_Norm[3] <= 0) && (CP_Norm[4] < 0)) && (
       CP_Norm[5] > 0)) ) {
      // Fell off at filament FRONT
      *coff_f += 1.0; // (+END)
      *Fil_ends_FLAG = -1.0;
    } else if ( (((CP_Norm[3] < 0) && (CP_Norm[4] < 0) && (
       CP_Norm[5] < 0)) && ((CP_Norm[0] >= 0) && (CP_Norm[1]
       >= 0)) && (CP_Norm[2] < 0)) ||
     (((CP_Norm[3] > 0) && (CP_Norm[4] > 0) && (CP_Norm[5] >
        0)) && ((CP_Norm[0] <= 0) && (CP_Norm[1] > 0)) && (
        CP_Norm[2] > 0)) ||
     (((CP_Norm[3] < 0) && (CP_Norm[4] < 0) && (CP_Norm[5] <
        0)) && ((CP_Norm[0] >= 0) && (CP_Norm[1] < 0)) && (
        CP_Norm[2] < 0)) ) {
      // Fell off at filament BACK
      *coff_b += 1.0; // (-END)
      *Fil_ends_FLAG = -1.0;
    }
  }
  /* -----------------------------------------*/

}
```

B.3.4 Net_Shrink_Grow.c

This program is used in the main to calculate filament dynamics and update new positions for the network.

```
// Net_Shrink_Grow.c: Imtiaz Ali
// Description: Function to shrink or grow filament network
// compiler: GCC 8.1
// last modified: 05/01/2021
// Input: or = outer cell radius, ir = inner cell radius,
   xcent = cell x center, ycent = cell y center
```

```
//        filNet_M[fil_#][0] = r1, filNet_M[fil_#][1] = theta,
    filNet_M[fil_#][2] = alpha, filNet_M[fil_#][3] = p
//        filEnds_M[fil_#][0] = x1, filEnds_M[fil_#][1] = x2,
   filEnds_M[fil_#][2] = y1, filEnds_M[fil_#][3] = y2
//        Grow_true = 1 (Defaulted to grow in main function)

#include "Net_Shrink_Grow.h"
#define M_PI 3.14159265358979323846

// ########################################### //
// ########################################### //

// NOTE: Closer filament is to edge (inner or outer), the more
    energy it needs to grow/shrink
// Key assumption: We do not let the filaments disappear! (
   Experimentally this will need to be done)

// Positive polarity points to positive end of filament:
  //  -> In our case, the way the equations are setup, this
     implies positive end is the ending side.
  //  -> The side we work on is independent of the motor we
     use.

// For Microtubule, need to use its polarity (p_0) to find its
    positive end.
  // -> If p_0 > 0, (radially outward) work with ending side (
     closer to cell wall)
  // -> If p_0 < 0, (radially inward) work with beginning side
      (closer to cell nucleus)
  // Kinesin move towards postive end of MT
  // Dynein move towards negative end of MT

// For Actin, both ends must be used if applicable.
  // Myosein move towards positive end of Actin -> Similar to
     Kinesin
  // This will be a little trickier to implement
```

```
// How to pick which side of filament is the ending side
   depending on the polarity
    // POSITIVE POLARITY (p_0 > 0) (Ending side is pointing
       Radially Outward to Cell Wall/Membrane)
    // Beginning Side (x1,y1)
       ---------------------------------- Ending Side (x2,y2)

    // NEGATIVE POLARITY (p_0 < 0) (Ending side is pointing
       Radially Inward to Cell Nucleus)
    // Ending Side (x1,y1) ----------------------------------
        Beginning Side (x2,y2)

// Microtubule shrink(seconds) much faster than they grow(
   minutes) -> ONLY positive end shrinks/grows for MT.
// Actin filaments shrink/grow from both ends.

// kinesin 0.2microns/s to 1microns/s with each step of 8 nm
   long. With range of t_fil = 8 * 10^-3 to 40 * 10^-3 seconds
// -> motor stepping time range is 0.8 to 4 seconds within t =
    100 to enter this code
// -> step depolimerization, assymetry is 5 to 20 -> 20
   percent growth is 2microns per minute =-> 2.5 percent
// For dynein, polarity does change depending on what alpha/
   beta-tubulin dimer it attaches too

// ########################################### //
// ########################################### //

void Net_SG_MICROTUBULE(size_t row, size_t colm, float Net_Fil
   [row][colm], float Net_Ends[row][colm], float or,
 float ir, float xcent, float ycent, int Grow_true, float DT,
    float V_G_MT, float V_S_MT, float min_f_length, float
    fil_buffer) {

  // ########################################### //
```

```
// ########################################### //
// Part 1: Declaration, Initialization, Assignment

// Original filament variables -> (lr = Filaments current
   length)
float x1_0, x2_0, y1_0, y2_0, r1_0, theta_0, alpha_0, p_0,
   lr, r2_0;

// New updated variables declaration -> (lr_n = Filaments
   new length)
float df_1, x_n, y_n, r_n, lr_n, theta_1, alpha_1, d_theta,
   gamma_1, lr_temp;
float d_change = fil_buffer; // Closest distance to inner or
    outer regions (microns)
float min_fil_length = min_f_length; // Smallest filament
   length (microns) -> Filament never disappears
float max_fil_length = 2*sqrt(pow(or,2) - pow(ir,2)); //
   Largest filament length (microns) -> (Max Cord Length) =
   2*sqrt(or^2-ir^2)

// Variables for staying within region (I don't use these in
    the region check anymore) (LEAVE FOR NOW)
// Inner region
float x_neg_min = xcent + -ir - d_change; // Nucleus
   negative max x-pos -> x1 < x_neg_min
float x_pos_min = xcent + ir + d_change; // Nucleus positive
    max x-pos  -> x1 > x_pos_min
float y_neg_min = ycent + -ir - d_change; // Nucleus
   negative max y-pos -> y1 < y_neg_min
float y_pos_min = ycent + ir + d_change; // Nucleus positive
    max y-pos  -> y1 > y_pos_min
// Outer Region
float x_neg_max = xcent + -or + d_change; // Cell negative
   max x-pos -> x2 > x_neg_max
float x_pos_max = xcent + or - d_change; // Cell positive
   max x-pos  -> x2 < x_pos_max
```

```
float y_neg_max = ycent + -or + d_change; // Cell negative
   max y-pos -> y2 > y_neg_max
float y_pos_max = ycent + or - d_change; // Cell positive
   max y-pos  -> y2 < y_pos_max

// Boolean for modifying beginning or ending side
int MOD_side; // Modify (x1, y1) = 0, Modify (x2,y2) = 1
float rand_n = (float)rand()/RAND_MAX; // Use this for
   Myosein-Actin Networks

// Choosing either shrinking, growing or do nothing to the
   filament
int shrink_start; // Flag for filament shrinking
int growth_start = Grow_true; // Grow = 1, Shrink = 0,
   Nothing = -1

// Set declared variables values -> (Filaments polarity
   never changes)
x1_0 = Net_Ends[row][0]; x2_0 = Net_Ends[row][1]; y1_0 =
   Net_Ends[row][2]; y2_0 = Net_Ends[row][3];
r1_0 = Net_Fil[row][0]; theta_0 = Net_Fil[row][1]; alpha_0 =
    Net_Fil[row][2]; p_0 = Net_Fil[row][3];

// ########################################### //
// ########################################### //
// Part 2: Current length calculation, Pick filament side to
    modify, Choose Grow, Shrink or do nothing

// Calculate filaments current length
lr = sqrt(pow((x2_0 - x1_0),2) + pow((y2_0 - y1_0),2));

// Pick which filament side we work on => Unique for
   MICROTUBULE
if (p_0 < 0) {
  // Negative Polarity
```

```
  MOD_side = 0; // Work with beginning side of filament =>
     x1, y1
} else {
  // Positive Polarity
  MOD_side = 1; // Work with ending side of filament => x2,
     y2
}

// Pick if we grow filament, shrink or do nothing to
   filament
if (growth_start == 1) {
  // Grow filament
  shrink_start = 0; // False -> Do not shrink, GROW
  // lr_n = lr * 1.10; // Microtubule Growth Filament 2.5% (
     microns)
  lr_n = lr + V_G_MT*DT; // New filament length ->
     Microtubule Growth Filament 2.5% (microns)
  // Check we didn't grow above max length
  if (lr_n > max_fil_length) {
    lr_n = max_fil_length - 0.05;
  }
} else if (growth_start == 0) {
  // Shrink filament
  shrink_start = 1; // True -> Do not grow, SHRINK
  lr_n = lr - V_S_MT*DT; // New filament length ->
     Microtubule Shrinking filament 20% (microns)
  // Check we didn't shrink below min length
  if (lr_n < min_fil_length) {
    lr_n = min_fil_length + 0.05;
  }
} else {
  // Do nothing to filament
  shrink_start = -1; // N/A -> Do Nothing
  lr_n = lr; // Keep Microtubules original length
}
```

```
// ########################################### //
// ########################################### //
// Part 3: Choosing to modify or not to modify filament if
   sufficient system requirement are met

// Start modification if requirments are met:
if ( ((lr < min_fil_length) && (shrink_start == 1)) || ((lr
   > max_fil_length) && (shrink_start == 0)) ||
  (shrink_start == -1) ) {
  // ########################################### //
  // ########################################### //
  // Part 3A: Do not modify filament because system
     requirements did not meet

  // Store original filament points
  Net_Ends[row][0] = x1_0; Net_Ends[row][2] = y1_0; //
     Original beginning side points
  Net_Ends[row][1] = x2_0; Net_Ends[row][3] = y2_0; //
     Original ending side points

} else {
  // Start filament modification

  // ########################################### //
  // ########################################### //
  // Part 3B: Start modification of filament

  // Condition statement for beginning side -> shrink, grow
     or nothing
  if (MOD_side == 0) {
    // ########################################### //
    // ########################################### //
    // Part 3B_1: Start modification on beginning side of
       filament
    //            -> Recalculate (x1,y1,r1,theta,alpha)
```

```
//            -> Don't recalculate (x2,y2,r2,beta=theta+
   alpha,GAMMA=theta+gamma)

// Calculate new beginning position for filament
x_n = x2_0 - lr_n * cos(theta_0 + alpha_0); // New x1
   location
y_n = y2_0 - lr_n * sin(theta_0 + alpha_0); // New y1
   location
r_n = sqrt(pow((x_n - xcent),2) + pow((y_n - ycent),2));
    // New beginning distance from center of cell r_1
theta_1 = atan((y_n - ycent) / (x_n - xcent)); // New
   CCW angle to filaments start point (Radians)
alpha_1 = theta_0 + alpha_0 - theta_1; // New difference
    angle between filament end point (Radians)

// Calculate filaments new current length
lr_temp = sqrt(pow((x2_0 - x_n),2) + pow((y2_0 - y_n),2)
   );

// Check if within proper region
// Condition check if new (r_1) is:
// -> Inside the nucleus, outside the cell, smaller than
    min length or larger than max length
if ( ((r_n < (ir + d_change)) && (shrink_start == 0)) ||
    ((r_n > (or - d_change)) && (shrink_start == 0)) ||
   (lr_temp < min_fil_length) ||
 (lr_temp > max_fil_length) ) {
  // Re-calculate new postions so its within region
  // lr_temp = lr * 0.90; // Let it shrink
  lr_temp = lr; // Do nothing
  x_n = x2_0 - lr_temp * cos(theta_0 + alpha_0);
  y_n = y2_0 - lr_temp * sin(theta_0 + alpha_0);
  theta_1 = atan((y_n - ycent) / (x_n - xcent)); // New
     CCW angle to filaments start point (Radians)
```

```
      alpha_1 = theta_0 + alpha_0 - theta_1; // New
         difference angle between filament end point (
         Radians)
    } // End condition to keep beginning side outside inner
       part of cell (nucleus)

    // Store filaments new beginning position
    Net_Ends[row][0] = x_n; Net_Ends[row][2] = y_n; // New
       beginning side points
    Net_Ends[row][1] = x2_0; Net_Ends[row][3] = y2_0; //
       Original ending side points
    Net_Fil[row][0] = r_n; Net_Fil[row][1] = theta_1;
       Net_Fil[row][2] = alpha_1; // New r1, theta, alpha
    Net_Fil[row][3] = p_0; // Filament polarity never
       changes

  } // End shrinking/growing beginning side of filament

  // Condition statement for ending side -> shrink, grow or
     nothing
  if (MOD_side == 1) {
    // ############################################ //
    // ############################################ //
    // Part 3B_2: Start modification on ending side of
       filament
    //            -> Recalculate (x2,y2,r2)
    //            -> Don't recalculate (x1,y1,r1,beta=theta+
       alpha)

    // Calculate new ending position for filament
    x_n = x1_0 + lr_n * cos(theta_0 + alpha_0); // New x2
       location
    y_n = y1_0 + lr_n * sin(theta_0 + alpha_0); // New y2
       location
    r_n = sqrt(pow((x_n - xcent),2) + pow((y_n - ycent),2));
        // New ending distance from center of cell r_2
```

```
    // Calculate filaments new current length
    lr_temp = sqrt(pow((x_n - x1_0),2) + pow((y_n - y1_0),2)
       );

    // Check if within proper region
    // Condition check if new (r_2) is:
    // -> Outside the cell, inside the nucleus, smaller than
        min length or larger than max length
    if ( ((r_n > (or - d_change)) && (shrink_start == 0)) ||
        ((r_n < (ir + d_change)) && (shrink_start == 0)) ||
       (lr_temp < min_fil_length) ||
     (lr_temp > max_fil_length) ) {
      // Re-calculate new postions so its within region
      // lr_temp = lr * 0.90; // Let it shrink
      lr_temp = lr; // Do nothing
      x_n = x1_0 + lr_temp * cos(theta_0 + alpha_0);
      y_n = y1_0 + lr_temp * sin(theta_0 + alpha_0);
    } // End condition to keep ending side inside outer part
        of cell (cell wall)

    // Store filaments new ending position
    Net_Ends[row][0] = x1_0; Net_Ends[row][2] = y1_0; //
       Original beginning side points
    Net_Ends[row][1] = x_n; Net_Ends[row][3] = y_n; // New
       ending side points
    Net_Fil[row][0] = r1_0; Net_Fil[row][1] = theta_0;
       Net_Fil[row][2] = alpha_0; // Original r1, theta,
       alpha
    Net_Fil[row][3] = p_0; // Filament polarity never
       changes

  } // End shrinking/growing ending side of filament

} // End modying filament if system requirements are met
// ########################################### //
```

```
  // ########################################### //
  // Part 3: End

}

void Net_SG_ACTIN(size_t row, size_t colm, float Net_Fil[row][
   colm], float Net_Ends[row][colm], float or,
 float ir, float xcent, float ycent, int Grow_true, int
    Actin_side, float DT, float V_G_ACTIN, float V_S_ACTIN,
    float min_f_length, float fil_buffer) {

  // ########################################### //
  // ########################################### //
  // Part 1: Declaration, Initialization, Assignment

  // Original filament variables -> (lr = Filaments current
     length)
  float x1_0, x2_0, y1_0, y2_0, r1_0, theta_0, alpha_0, p_0,
     lr, r2_0;

  // New updated variables declaration -> (lr_n = Filaments
     new length)
  float df_1, x_n, y_n, r_n, lr_n, theta_1, alpha_1, d_theta,
     gamma_1, lr_temp;
  float d_change = fil_buffer; // Closest distance to inner or
      outer regions (microns)
  float min_fil_length = min_f_length; // Smallest filament
     length (microns) -> Filament never disappears
  float max_fil_length = 2*sqrt(pow(or,2) - pow(ir,2)); //
     Largest filament length (microns) -> (Max Cord Length) =
     2*sqrt(or^2-ir^2)

  // Actin ONLY - Filament Side
  int A_side = Actin_side; // No side = -1, Beginning side =
     0, Ending side = 1
```

```
// Boolean for modifying beginning or ending side
int MOD_side; // Modify (x1, y1) = 0, Modify (x2,y2) = 1
float rand_n = (float)rand()/RAND_MAX; // Use this for
   Myosein-Actin Networks

// Choosing either shrinking, growing or do nothing to the
   filament
int shrink_start; // Flag for filament shrinking
int growth_start = Grow_true; // Grow = 1, Shrink = 0,
   Nothing = -1

// Set declared variables values -> (Filaments polarity
   never changes)
x1_0 = Net_Ends[row][0]; x2_0 = Net_Ends[row][1]; y1_0 =
   Net_Ends[row][2]; y2_0 = Net_Ends[row][3];
r1_0 = Net_Fil[row][0]; theta_0 = Net_Fil[row][1]; alpha_0 =
    Net_Fil[row][2]; p_0 = Net_Fil[row][3];

// ########################################### //
// ########################################### //
// Part 2: Current length calculation, Pick filament side to
    modify, Choose Grow, Shrink or do nothing

// Calculate filaments current length
lr = sqrt(pow((x2_0 - x1_0),2) + pow((y2_0 - y1_0),2));

// Pick which filament side we work on => Unique for ACTIN
if (p_0 < 0) {
  // Negative Polarity
  if (A_side == 0) {
    MOD_side = 1; // Work with beginning side of filament =>
        x2, y2
  } else {
    MOD_side = 0; // Work with ending side of filament => x1
       . y1
  }
```

```
} else {
  // Positive Polarity
  if (A_side == 1) {
    MOD_side = 1; // Work with ending side of filament => x2
       , y2
  }else {
    MOD_side = 0; // Work with beginning side of filament =>
        x1, y1
  }
}

// Pick if we grow, shrink, or do nothing to filament
if (growth_start == 1) {
  // Grow filament
  shrink_start = 0; // False -> Do not shrink, GROW
  // lr_n = lr * 1.0; // Actin Growth Filament 2.5% (microns
     )
  lr_n = lr + V_G_ACTIN*DT; // New filament length -> Actin
     Growth Filament 2.5% (microns)
  // Check we didn't grow above max length
  if (lr_n > max_fil_length) {
    lr_n = max_fil_length - 0.05;
  }
} else if (growth_start == 0) {
  // Shrink filament
  shrink_start = 1; // True -> Do not grow, SHRINK
  lr_n = lr - V_S_ACTIN*DT; // New filament length -> Actin
     Shrinking filament 20% (microns)
  // Check we didn't shrink below min length
  if (lr_n < min_fil_length) {
    lr_n = min_fil_length + 0.05;
  }
} else {
  // Do nothing to filament
  shrink_start = -1; // N/A -> Do Nothing
  lr_n = lr; // Keep Actin original length
```

```
}

// ########################################### //
// ########################################### //
// Part 3: Choosing to modify or not to modify filament if
   sufficient system requirement are met

// Start modification if requirments are met:
if ( ((lr < min_fil_length) && (shrink_start == 1)) || ((lr
   > max_fil_length) && (shrink_start == 0)) ||
  (shrink_start == -1) ) {
  // ########################################### //
  // ########################################### //
  // Part 3A: Do not modify filament because system
     requirements did not meet

  // Store original filament points
  Net_Ends[row][0] = x1_0; Net_Ends[row][2] = y1_0; //
     Original beginning side points
  Net_Ends[row][1] = x2_0; Net_Ends[row][3] = y2_0; //
     Original ending side points

} else {
  // Start filament modification

  // ########################################### //
  // ########################################### //
  // Part 3B: Start modification of filament

  // Condition statement for beginning side -> shrink, grow
     or nothing
  if (MOD_side == 0) {
    // ########################################### //
    // ########################################### //
    // Part 3B_1: Start modification on beginning side of
       filament
```

```
//            -> Recalculate (x1,y1,r1,theta,alpha)
//            -> Don't recalculate (x2,y2,r2,beta=theta+
   alpha,GAMMA=theta+gamma)

// Calculate new beginning position for filament
x_n = x2_0 - lr_n * cos(theta_0 + alpha_0); // New x1
   location
y_n = y2_0 - lr_n * sin(theta_0 + alpha_0); // New y1
   location
r_n = sqrt(pow((x_n - xcent),2) + pow((y_n - ycent),2));
    // New beginning distance from center of cell r_1
theta_1 = atan((y_n - ycent) / (x_n - xcent)); // New
   CCW angle to filaments start point (Radians)
alpha_1 = theta_0 + alpha_0 - theta_1; // New difference
    angle between filament end point (Radians)

// Calculate filaments new current length
lr_temp = sqrt(pow((x2_0 - x_n),2) + pow((y2_0 - y_n),2)
   );

// Check if within proper region
// Condition check if new (r_1) is:
// -> Inside the nucleus, outside the cell, smaller than
    min length or larger than max length
if ( ((r_n < (ir + d_change)) && (shrink_start == 0)) ||
    ((r_n > (or - d_change)) && (shrink_start == 0)) ||
   (lr_temp < min_fil_length) ||
 (lr_temp > max_fil_length) ) {
  // Re-calculate new postions so its within region
  // lr_temp = lr * 0.90; // Let it shrink
  lr_temp = lr; // Do nothing
  x_n = x2_0 - lr_temp * cos(theta_0 + alpha_0);
  y_n = y2_0 - lr_temp * sin(theta_0 + alpha_0);
  theta_1 = atan((y_n - ycent) / (x_n - xcent)); // New
     CCW angle to filaments start point (Radians)
```

```
      alpha_1 = theta_0 + alpha_0 - theta_1; // New
         difference angle between filament end point (
         Radians)
    } // End condition to keep beginning side outside inner
       part of cell (nucleus)

    // Store filaments new beginning position
    Net_Ends[row][0] = x_n; Net_Ends[row][2] = y_n; // New
       beginning side points
    Net_Ends[row][1] = x2_0; Net_Ends[row][3] = y2_0; //
       Original ending side points
    Net_Fil[row][0] = r_n; Net_Fil[row][1] = theta_1;
       Net_Fil[row][2] = alpha_1; // New r1, theta, alpha
    Net_Fil[row][3] = p_0; // Filament polarity never
       changes

  } // End shrinking/growing beginning side of filament

  // Condition statement for ending side -> shrink, grow or
     nothing
  if (MOD_side == 1) {
    // ############################################ //
    // ############################################ //
    // Part 3B_2: Start modification on ending side of
       filament
    //            -> Recalculate (x2,y2,r2)
    //            -> Don't recalculate (x1,y1,r1,beta=theta+
       alpha)

    // Calculate new ending position for filament
    x_n = x1_0 + lr_n * cos(theta_0 + alpha_0); // New x2
       location
    y_n = y1_0 + lr_n * sin(theta_0 + alpha_0); // New y2
       location
    r_n = sqrt(pow((x_n - xcent),2) + pow((y_n - ycent),2));
        // New ending distance from center of cell r_2
```

```
    // Calculate filaments new current length
    lr_temp = sqrt(pow((x_n - x1_0),2) + pow((y_n - y1_0),2)
       );

    // Check if within proper region
    // Condition check if new (r_2) is:
    // -> Outside the cell, inside the nucleus, smaller than
        min length or larger than max length
    if ( ((r_n > (or - d_change)) && (shrink_start == 0)) ||
        ((r_n < (ir + d_change)) && (shrink_start == 0)) ||
       (lr_temp < min_fil_length) ||
     (lr_temp > max_fil_length) ) {
      // Re-calculate new postions so its within region
      // lr_temp = lr * 0.90; // Let it shrink
      lr_temp = lr; // Do nothing
      x_n = x1_0 + lr_temp * cos(theta_0 + alpha_0);
      y_n = y1_0 + lr_temp * sin(theta_0 + alpha_0);
    } // End condition to keep ending side inside outer part
        of cell (cell wall)

    // Store filaments new ending position
    Net_Ends[row][0] = x1_0; Net_Ends[row][2] = y1_0; //
       Original beginning side points
    Net_Ends[row][1] = x_n; Net_Ends[row][3] = y_n; // New
       ending side points
    Net_Fil[row][0] = r1_0; Net_Fil[row][1] = theta_0;
       Net_Fil[row][2] = alpha_0; // Original r1, theta,
       alpha
    Net_Fil[row][3] = p_0; // Filament polarity never
       changes

  } // End shrinking/growing ending side of filament

} // End modying filament if system requirements are met
// ########################################### //
```

```
  // ############################################ //
  // Part 3: End

}
```

Bibliography

[1] Rossana Rojas Molina, Susanne Liese, and Andreas Carlson. Diffusion on membrane domes, tubes, and pearling structures. *Biophysical Journal*, 120(3):424–431, 2021.

[2] Molecular expressions cell biology: Microtubules. `https://micro.magnet.fsu.edu/cells/microtubules/microtubules.html`. Accessed: October 06, 2021.

[3] David Ando, Nickolay Korabel, Kerwyn Casey Huang, and Ajay Gopinathan. Cytoskeletal network morphology regulates intracellular transport dynamics. *Biophysical journal*, 109(8):1574–1582, 2015.

[4] Edward A Codling, Michael J Plank, and Simon Benhamou. Random walk models in biology. *Journal of the Royal society interface*, 5(25):813–834, 2008.

[5] Marina Sidortsov, Yakov Morgenstern, and Avraham Be'er. Role of tumbling in bacterial swarming. *Physical Review E*, 96(2):022407, 2017.

[6] Hideki Takayasu. Stable distribution and levy process in fractal turbulence. *Progress of theoretical physics*, 72(3):471–479, 1984.

[7] Irina Semenova, Anton Burakov, Neda Berardone, Ilya Zaliapin, Boris Slepchenko, Tatyana Svitkina, Anna Kashina, and Vladimir Rodionov. Actin dynamics is essential for myosin-based transport of membrane organelles. *Current Biology*, 18(20):1581–1586, 2008.

[8] George H Weiss and Robert J Rubin. Random walks: theory and selected applications. *Adv. Chem. Phys*, 52:363–505, 1983.

[9] Howard C Berg. *Random walks in biology*. Princeton University Press, 2018.

[10] Thanu Padmanabhan. More on random walks: Circuits and a tired drunkard. In *Sleeping Beauties in Theoretical Physics*, pages 259–268. Springer, 2015.

[11] Daniel Ben-Avraham and Shlomo Havlin. *Diffusion and reactions in fractals and disordered systems*. Cambridge university press, 2000.

[12] G Zumofen, J Klafter, and MF Shlesinger. Anomalous diffusion (lecture notes in physics, vol. 519) ed. a. pekalski and k. sznajd-weron, 1999.

[13] Joseph Klafter, Michael F Shlesinger, and Gert Zumofen. Beyond brownian motion. *Physics today*, 49(2):33–39, 1996.

[14] Benjamin M Regner, Dejan Vučinić, Cristina Domnisoru, Thomas M Bartol, Martin W Hetzer, Daniel M Tartakovsky, and Terrence J Sejnowski. Anomalous diffusion of single particles in cytoplasm. *Biophysical journal*, 104(8):1652–1660, 2013.

[15] Adal Sabri, Xinran Xu, Diego Krapf, and Matthias Weiss. Elucidating the origin of heterogeneous anomalous diffusion in the cytoplasm of mammalian cells. *Physical Review Letters*, 125(5):058101, 2020.

[16] Jennifer L Ross, M Yusuf Ali, and David M Warshaw. Cargo transport: molecular motors navigate a complex cytoskeleton. *Current opinion in cell biology*, 20(1):41–47, 2008.

[17] Bryan Maelfeyt, SM Ali Tabei, and Ajay Gopinathan. Anomalous intracellular transport phases depend on cytoskeletal network features. *Physical Review E*, 99(6):062404, 2019.

[18] C Loverdo, O Bénichou, M Moreau, and R Voituriez. Enhanced reaction kinetics in biological cells. *Nature physics*, 4(2):134, 2008.

[19] J Tailleur and ME Cates. Statistical mechanics of interacting run-and-tumble bacteria. *Physical review letters*, 100(21):218103, 2008.

[20] V Zaburdaev, S Denisov, and Joseph Klafter. Levy walks. *Rev. Mod. Physics*, 87(2):483, 2015.

[21] Gandhimohan M Viswanathan, Marcos GE Da Luz, Ernesto P Raposo, and H Eugene Stanley. *The physics of foraging: an introduction to random searches and biological encounters*. Cambridge University Press, 2011.

[22] Gandhimohan M Viswanathan, V Afanasyev, SV Buldyrev, EJ Murphy, PA Prince, and H Eugene Stanley. Lévy flight search patterns of wandering albatrosses. *Nature*, 381(6581):413, 1996.

[23] Gabriel Ramos-Fernández, José L Mateos, Octavio Miramontes, Germinal Cocho, Hernán Larralde, and Barbara Ayala-Orozco. Lévy walk patterns in the foraging movements of spider monkeys (ateles geoffroyi). *Behavioral ecology and Sociobiology*, 55(3):223–230, 2004.

[24] RPD Atkinson, CJ Rhodes, DW Macdonald, and RM Anderson. Scale-free dynamics in the movement patterns of jackals. *Oikos*, 98(1):134–140, 2002.

[25] Gil Ariel, Amit Rabani, Sivan Benisty, Jonathan D Partridge, Rasika M Harshey, and Avraham Be'Er. Swarming bacteria migrate by lévy walk. *Nature communications*, 6:8396, 2015.

[26] Felix Höfling and Thomas Franosch. Anomalous transport in the crowded world of biological cells. *Reports on Progress in Physics*, 76(4):046602, 2013.

[27] Takahiro K Fujiwara, Kokoro Iwasawa, Ziya Kalay, Taka A Tsunoyama, Yusuke Watanabe, Yasuhiro M Umemura, Hideji Murakoshi, Kenichi GN Suzuki, Yuri L Nemoto, Nobuhiro Morone, et al. Confined diffusion of transmembrane proteins and lipids induced by the same actin meshwork lining the plasma membrane. *Molecular biology of the cell*, 27(7):1101–1119, 2016.

[28] Jordi Faraudo. Diffusion equation on curved surfaces. i. theory and application to biological membranes. *The Journal of chemical physics*, 116(13):5831–5841, 2002.

[29] Daniel Coombs, Ronny Straube, and Michael Ward. Diffusion on a sphere with localized traps: Mean first passage time, eigenvalue asymptotics, and fekete points. *SIAM Journal on Applied Mathematics*, 70(1):302–332, 2009.

[30] F Debbasch. A diffusion process in curved space–time. *Journal of mathematical physics*, 45(7):2744–2760, 2004.

[31] Ramón Castañeda-Priego, Pavel Castro-Villarreal, Sendic Estrada-Jiménez, and José Miguel Méndez-Alcaraz. Brownian motion of free particles on curved surfaces. *arXiv preprint arXiv:1211.5799*, 2012.

[32] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguná. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, 2010.

[33] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[34] Benjamin Paul Chamberlain, James Clough, and Marc Peter Deisenroth. Neural embeddings of graphs in hyperbolic space. *arXiv preprint arXiv:1705.10359*, 2017.

[35] Marián Boguná, Fragkiskos Papadopoulos, and Dmitri Krioukov. Sustaining the internet with hyperbolic mapping. *Nature communications*, 1:62, 2010.

[36] James R Clough and Tim S Evans. What is the dimension of citation space? *Physica A: Statistical Mechanics and its Applications*, 448:235–247, 2016.

[37] James R Clough, Jamie Gollings, Tamar V Loach, and Tim S Evans. Transitive reduction of citation networks. *Journal of Complex Networks*, 3(2):189–203, 2015.

[38] Rastko Sknepnek and Silke Henkes. Active swarms on a sphere. *Physical Review E*, 91(2):022306, 2015.

[39] Nihad E Daidzic. Long and short-range air navigation on spherical earth. *International Journal of Aviation, Aeronautics, and Aerospace*, 4(1):2, 2017.

[40] A Blumen, G Zumofen, and J Klafter. Transport aspects in anomalous diffusion: Lévy walks. *Physical Review A*, 40(7):3964, 1989.

[41] Pavel Castro-Villarreal. Brownian motion meets riemann curvature. *Journal of Statistical Mechanics: Theory and Experiment*, 2010(08):P08006, 2010.

[42] John Ratcliffe. *Foundations of hyperbolic manifolds*, volume 149. Springer Science & Business Media, 2006.

[43] Dirk Jan Struik. *Lectures on classical differential geometry*. Courier Corporation, 1961.

[44] N. Hitchin. Geometry of surfaces, 2013.

[45] Izrail Moiseevitch Gelfand, Richard A Silverman, et al. *Calculus of variations*. Courier Corporation, 2000.

[46] L Meister and H Schaeben. A concise quaternion geometry of rotations. *Mathematical Methods in the Applied Sciences*, 28(1):101–126, 2005.

[47] SM Ali Tabei, Stanislav Burov, Hee Y Kim, Andrey Kuznetsov, Toan Huynh, Justin Jureller, Louis H Philipson, Aaron R Dinner, and Norbert F Scherer. Intracellular transport of insulin granules is a subordinated random walk. *Proceedings of the National Academy of Sciences*, 110(13):4911–4916, 2013.

[48] William M Saxton and Peter J Hollenbeck. The axonal transport of mitochondria. *Journal of cell science*, 125(9):2095–2104, 2012.

[49] Ajay Gopinathan, Kun-Chun Lee, Jennifer M Schwarz, and Andrea J Liu. Branching, capping, and severing in dynamic actin structures. *Physical review letters*, 99(5):058103, 2007.

[50] P Bleicher, A Sciortino, and AR Bausch. The dynamics of actin network turnover is self-organized by a growth-depletion feedback. *Scientific reports*, 10(1):1–11, 2020.

[51] Ron Milo and Rob Phillips. *Cell biology by the numbers*. Garland Science, 2015.

[52] Peter Hänggi, Peter Talkner, and Michal Borkovec. Reaction-rate theory: fifty years after kramers. *Reviews of modern physics*, 62(2):251, 1990.