UNIVERSITY OF CALIFORNIA, MERCED

The Study of Intracellular Transport From the Perspective of an Explicit Cytoskeletal Network Geometry Using Simulation and Numerical Integration Techniques

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

Physics

by

Bryan Maelfeyt

Committee in charge:

Professor Linda Hirst, Chair
Professor Ajay Gopinathan
Professor Jing Xu
Professor Bin Liu

May 2019

The dissertation of Bryan Maelfeyt, titled The Study of Intracellular Transport From the Perspective of an Explicit Cytoskeletal Network Geometry Using Simulation and Numerical Integration Techniques, is approved, and it is acceptable in quality and form for publication.

(Professor Ajay Gopinathan) Principal Adviser	Date
(Professor Linda Hirst) Committee Chair	Date
(Professor Jing Xu) Committee Member	Date
(Professor Bin Liu) Committee Member	Date

University of California, Merced
2019

This dissertation is dedicated to the memory of my grandfather, Leo Maelfeyt.

Acknowledgments

I want to thank my advisor, Ajay Gopinathan and my committee members for their guidance. I would like to thank Ali Tabei, Jenny Ross, David Quint, and Niranjan Sparpangala for their help with different parts of my research. Finally, I want to thank my friends, family, and my finacée, Alison Huff for their emotional support. This research has been funded by the following sources:

- National Science Foundation (NSF) grants DMS-1616926 and EF-1038697
- NSF-CREST: Center for Cellular and Bio-molecular Machines (CCBM) at UC Merced (NSF-HRD-1547848)
- James S. McDonnell Foundation 21st Century Science Initiative Award

Bryan Maelfeyt
Curriculum Vitae

Contact:
bmaelfeyt@ucmerced.edu
bryjmaelfeyt@gmail.com

Current Employer:
University of California, Merced
5200 Lake Rd
Merced, CA 95343

Education:

Current Institution:
UC Merced Physics Graduate Group (Ph. D program, began 2013)
Advisor: Ajay Gopinathan
GPA: 4.00
Passed Preliminary Examination
Passed Qualifying Examination
Expected graduation date: May 2019

Previous Institution:
B. S. in Physics (UC Davis, graduated in June 2012)

Employment:
Teaching Assistant (TA) for Quantum Mechanics II, (UC Merced, Spring 2019)
TA for Analytical Mechanics, (UC Merced, Spring 2019)
TA for Biophysics, (UC Merced, Fall 2018)
Graduate Student Researcher (GSR), Physics, UC Merced (Dr. Ajay Gopinathan, Summer 2018)
TA for Calculus I (UC Merced, Fall 2017)
GSR in Physics, UC Merced (Dr. Ajay Gopinathan, Fall 2016)
TA for Physics I (UC Merced, Summer 2016)
GSR in Physics at UC Merced (Dr. Kevin Mitchell, Summer 2015 – Spring 2016)
TA for Calculus I (UC Merced, Fall 2014 – Spring 2015)
GSR in Physics at UC Merced (Dr. Sayantani Ghosh, Summer 2014)
TA for Calculus II (UC Merced, Fall 2013 – Spring 2014)
Junior Research Specialist in Physics at UC Davis (Dr. Richard Scalettar, Fall 2012)

Research Skills:
Programming experience in C, C++, Python, and MATLAB
Data analysis experience in Python and MATLAB

Proficient in Microsoft Office and related software
Extensive use of LaTeX typesetting language

Research:

Molecular motor-driven intracellular transport (current work)
We've investigated how cytoskeletal network variability affects the efficiency of intracellular transport using numerical integrations and Monte Carlo simulations.

Topological analysis of volume-preserving maps (previous work)
We were able to extend the technique of homotopic lobe dynamics to quantify the complexity of three-dimensional, volume-preserving maps using the mathematical technique of symbolic dynamics.

Awards and Honors:
Best graduate student performance on the preliminary exam (UC Merced, 2013 - 2014)
Best graduate student performance in coursework (UC Merced, 2013 - 2014)
Graduated with honors (UC Davis, 2012)
Dean's Honors List (UC Davis, 2008 - 2012)

Memberships:
American Physical Society (APS), joined Fall 2013

Publications:

Bryan Maelfeyt, Spencer A. Smith, Kevin A. Mitchell, "Using invariant manifolds to construct symbolic dynamics for three-dimensional volume-preserving," SIAM Journal on Applied Dynamical Systems, 2017

Bryan Maelfeyt, S. M. Ali Tabei, Ajay Gopinathan, "Filament length drives anomalous transport over an explicit cytoskeletal network", In prep.

Bryan Maelfeyt, Ajay Gopinathan, "The effects of filament length and filament polarization on an evolving distribution of cargos", In prep.

Presentations:

APS March Meeting 2017: "Intracellular Transport of Cargo in a Sub-diffusive Environment over an Explicit Cytoskeletal Network"

APS Far West 2018: "Intracellular Transport is Sensitive to Filament Polarization"

Abstract

Intracellular transport in eukaryotic cells is a process in which cargo, carrying various materials and attached to molecular motors, moves around the cell. The cargos' transport consists of phases of passive, diffusion-based transport in the bulk cytoplasm and active, motor-driven transport along filaments that make up the cell's cytoskeleton. Because of it's role in the active phase of transport, the cytoskeletal geometry is an important factor. In this dissertation, we consider network parameters such as filament length, number, polarization direction, and location and examine their effect on the transport process. This can be achieved by computationally determining cargo transport through simulation and numerical analysis techniques.

We present this research by first demonstrating an approach that evolves a distribution of cargos in time using numerical integration. To do this, we use two coupled differential equations that enforce the distribution movement on and off filaments. An interesting finding here is that the distribution can become "trapped" at what we consider intermediate filament lengths.

Although we mostly use a simplified model where normal diffusion governs the passive phase of transport, we also consider the effects of incorporating anomalous subdiffusion in the bulk. This means that the entire transport process can be described as anomalously diffusive, with the active transport phase being superdiffusive and the passive transport phase being subdiffusive. One thing we found by taking this approach is that filament length, rather than filament number, has a greater influence on the domination of overall superdiffusive transport at relatively early times compared to the domination of subdiffusive transport at later times. We were able to extend this observation to model the biphasic release of insulin out of cells in which there is a large spike in insulin release, followed by a slower, more sustained release.

In the final chapter, we consider the possibility of cargos capable of switching to different filaments. If multiple motors are attached to a cargo, it can switch from one filament to another, provided one is nearby. In this phase of our research, we took real images of networks of microtubule bundles and extracted network parameters from them in order to run our simulations. We compared our simulation data, where cargos had different switching probabilities, with experimental data, where cargos had different numbers of motors, and were able to draw a correlation between cargo switching probability and motor number. The network images and the experimental data were provided by our collaborator, Professor Jennifer Ross at UMass, Amherst.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

The problem of transport has become widely studied at multiple length scales in biological systems. Many of these transport processes are stochastic, being represented as the time evolution of the probability that the system is in a certain state. This is especially the case for intracellular transport which occurs in nearly all eukaryotic cells.

In this environment multiple materials, including proteins, carbohydrates, nucleic acids, and lipids are tranported around the cell, to targets like the cell membrane, the nucleus, and the various organelles [1]. Because the transported material is sufficiently large (>nm) and transport distance is relatively far (>10 μm in eukaryotic cells compared to < 1 μm in prokaryotic cells where there is no active transport), reliance on diffusion alone to facilitate transport is not enough. These "large" materials, then, require some sort of active transport to aid their movement [2, 3]. As a result, intracellular transport is a combination of passive (diffusive) and active transport and is important in maintaining proper cellular function. Some examples of materials that rely on intracellular transport and, by extension, active transport, are vesicles containing proteins like insulin [4] and organelles like mitochondria [5].

In the cellular environment, active transport is accomplished through ATP-powered, molecular motor-driven transport along a complex cytoskeletal network consisting of polymers such as actin filaments and mircotubules [6]. Primarily, there are three kinds of molecular motors (myosin, kinesin, dynein) that transport cargos, with each using ATP to walk in a hand-over-hand type of motion upon their respective filament types [6]. Each motor is categorized by the type of filament on which they move, and the way in which they move upon them. Myosins tend to move toward what is considered the positive (+), polarized, end of actin filaments, which tend to form random networks within the cytoplasm [7, 8]. Kinesins and dyneins transport cargo along microtubules, which typically span across the cyoplasm from the nucleus to the cell membrane, with kinesins moving toward the (+) end, away from the nucleus. In contrast, dyneins move toward the (-) end of the microtubules [9,10]. Fig. 1.1a shows

an artist's depiction of kinesin motors walking on a cytoskeletal network composed of microtubules.

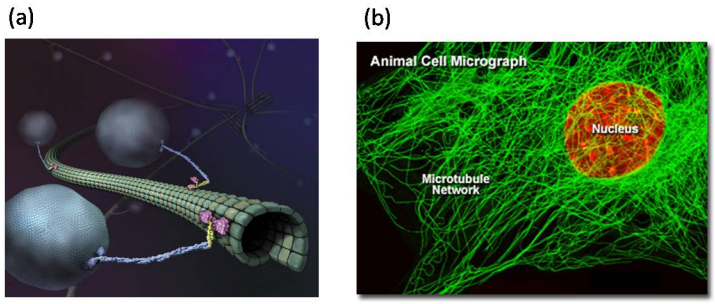

Figure 1.1: (a) Kinesin motors walking along a microtubule [11]. (b) An image of a microtubule network within an embryonic mouse cell [12].

The filaments form a network "mesh", consisting of microtubules providing transport tracks along long distances (can be greater than 25 μm in some neurons) and actin filaments providing transport over shorter distances (typically less than 1 μm). Cargos usually have multiple motor-binding sites providing for switching between filaments in the network [10]. Shown in Fig. 1.1b is a visual representation of a cytoskeletal network in an embryonic mouse cell.

The process of intracellular transport can then be thought of as occurring in two phases. Cargo alternates between motor-facilitated transport along a complex, dynamic cytoskeletal network consisting of mainly actin filaments and microtubules, and diffusive transport through the noisy, crowded environment that is the cytoplasm. Cargos, and the motors they're attached to, thus alternately bind and unbind from the filaments they walk on until they fall off, forcing the cargo to rely on diffusive transport for some time before attaching to another filament. This process continues until the cargo reaches its destination.

The process of cargo transport has been studied extensively from multiple perspectives including that of the molecular motors and their co-ordination [13–19] and features of the individual filaments they walk along [20–24]. More recently, there has been a lot of work, especially theoretical, focusing on the larger scales aspects of transport including the coupling of active and passive motion [4, 25–28], the role of geometric confinement [29, 30] and the geometry of the cytoskeletal network itself [24, 31–34].

The understanding of intracellular transport has important biological implications. For example, the breakdown of intracellular transport in neurons can lead to neurodegenerative diseases like Alzheimer's [35]. It is also the case that in the diabetic state, insulin granules' ability to be transported out of the cell is hindered, for reasons that are not fully understood [4]. The breakdown of the cytoskeletal network itself has been linked to various neurological diseases, immunodeficiency diseases, and development defects [36]. Studying intracellular transport from the perspective of the cytoskeletal network is also important because it is unique. Other transport models

have been attempted but have fallen short in giving a good description of intracellular transport processes. For example, in virtual network models, it is assumed that cargo randomly binds and unbinds from filaments, switching between phases of diffusion and ballistic motion of constant velocity [8,37]. One issue with this model is that the random binding and unbinding could happen anywhere within the system, as there is no explicit filament network considered in the model, meaning that homogeneity is assumed [32,38]. This is in contrast to the systems we use in our analyses where we use an explicit, heterogenious cytoskeletal network. Previous work in our group has found that trapping within certain regions of the cytoplasm can occur and that transport is quicker and more predictable when the filaments are placed closer to the starting position of cargos, polarity of the filaments points towards the target destination, there are more filaments with shorter lengths (as opposed to fewer filaments of greater length), and when binding and unbinding rates are optimized [31]. Due to findings like these, we believe that using a system that implements real, explicit filaments is vital to building up a good predictive model.

1.2 Overview

The results discussed here can be broken down into three parts. In general, we wish to characterize transport by evaluating first-passage time distributions (FPTDs) and mean first-passage times (MFPTs) as a function of the morphology of the cytoskeletal network. In this dissertation, we focus on developing new techniques to incorporate explicit filaments into numerical schemes and also to explore features of transport incorporating aspects that have not been accounted for before such as anomlous subdiffusion in the bulk cytoplasm and switching at filament intersections. Much of what is written here, particularly in chapters three and four, are parts of papers that are either in preparation or have been submitted.

1.2.1 Numerical Integration Techniques

We present a method to evolve a probability distribution of cargos in time. By doing this, we eliminate the noise inherent in the simulation of cargo movement and can examine broad cargo behavior at any instance in time. To use this method, we solve a set of coupled differential equations [3] that describe the time evolution of a cargo distrubution that is able to move on and off a filament network. We are able to see that for a sufficiently high fraction of filaments pointing towards the center of the cell, transport outward is slowed down and the distribution may become “trapped” in certain regions. One significant finding is that maximal trapping can be seen at intermediate filament lenghts. We explore these results more closely in chapter three.

1.2.2 Cargo Simulations Incorporating Anomalous Diffusion

Although most of the time, we simplify our model by considering normal diffusion within the bulk cytoplasm, technically, the entire intracellular transport process can be described by anomalous diffusion. The active transport phase can be considered superdiffusive and the passive transport phase within the bulk cytoplasm can be characterized as subdiffusive which has been shown experimentally [3, 39]. We evaluate the inter-play between superdiffusion and subdiffusion as characterized here. Ultimately, longer filaments help facilitate more superdiffusive transport. Superdiffusive transport turns out to dominate at earlier times. Another thing we were able to capture with our methodology here is the biphasic nature of insulin release out of the cell which is characterized by an initial spike in release, followed by one that is slow and sustained [4]. We explore this topic further in chapter four.

1.2.3 Cargo Simulations Using Networks Extracted from Images

We were able to extract networks from images of networks of microtuble bundles which were used in experiments that tracked cargo directly from our collaborator, Professor Jenny Ross. After running simulations using the extracted networks, we were able to compare simulation data with experimental data and draw a correlation between number of motors attached to individual cargos and the probability that a cargo will switch to another filament when it is near a filament intersection. We were able to find some correspondence of a switching probability of 0 with a one-motor cargo and a switching probability of 0.6 with a ten-motor cargo. One thing that is significant about what we have done here is that we have presented a way to take an image of a filament network and extract the explicit filaments that may be used in simulations or numerical integrations. This work is discussed in more detail in chapter five.

Chapter 2

Methods

In most of our analyses, we typically model a cargo moving within a eukaryotic cell as a random walker. Determining the time it takes for a random walker to reach its target destination, the first-passage time, provides a fundamental understanding of the particular random walker [37]. For multiple random walkers (the cargos in our model), the first-passage times we calculate can be used to construct a FPTD. The average of all of the first-passage times is MFPT. In our model, we will obtain FPTDs and MFPTs for cargos starting on the boundary of the nucleus (near the center of the cell) with their target destination being the cellular membrane (the cell's "outer boundary"). Both the MFPT and the FPTD can be used to quantify the effect the cell's cytoskeleton has on transport.

2.1 The system

In determining the MFPT and the FPTD, we will model both the *explicit simulation* of multiple cargos, and the *integration* of the time-evolution differential equations for a distribution of cargos. Simulations are easier to implement but numerical integrations give us more accurate answers for averaged quantities without requiring a lot of statistics. However, simulations of individual cargos can give us insight into the variance for a finite number of cargos and allow us to easily measure features of single trajectories for comparison with appropriate experiments. Thus, both have their advantages depending on the questions asked and the available data.

In our simulations, we use a model where we consider a simple eukaryotic cell containing a nucleus and a cell membrane. We also include a cytoskeletal network made up of a number of actin filaments of a given length and random orientations (Fig. 2.1) and sometimes include microtubules which usually span the entire cytoplasm of the cell, being typically much longer than the actin filaments [10]. the individual cargos (seen as red dots) start near the nucleus and then begin phases of passive and active transport, the latter of which being facilitated by molecular motors that walk along filaments, until they reach the outer cellular membrane. For now at least, we consider previously used parameters (cell radius of 10 μm, nuclear radius of 5 μm,

cargo radius of 100 nm, and motor speed of 1 μm/s) that are close to their real, physical quantities [31].

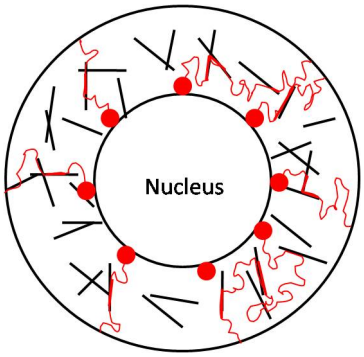

Figure 2.1: Our physical network system. Cargo (shown as red dots) starts on the nucleus then begins phases of passive and active transport until it reaches the outer cell membrane (the paths taken are shown as red lines)

2.2 Random walkers: explicit cargo simulations

In our simulations, we model individual cargos diffusing in the cytoplasm as random walkers. For a typical random walker, its net displacement is [40]:

$$\vec{r}(n) = \sum_{i=1}^{n} (a)\hat{e}_i \tag{2.1}$$

where $\hat{e}_i$ is a unit vector pointing in a direction that the random walker may move to at the next step. The next position that the walker might occupy is a distance a away.

After n time steps, a time t has passed where $t = n\tau$ and τ is the amount of time in one time step. The mean squared displacement as a function of the time t that has passed is then:

$$\langle r^2(t) \rangle = \frac{(a)^2 t}{\tau} \tag{2.2}$$

We can write this in terms of what is known as the diffusion constant, $D = (a)^2/(2d)\tau$ [40]

$$\langle r^2(n) \rangle = (2d)Dt \tag{2.3}$$

where d is the dimension of the lattice and, in our model, $D = 0.051$ μm/s^2 which is representative of a real cytoplasmic environment [31]. Eq. 2.3 means that the random walker undergoes Brownian motion, a diffusive process. Our result,

$$\langle r^2(t) \rangle \sim t \tag{2.4}$$

is a characteristic of diffusion.

We then model the passive transport of our cargo as undergoing diffusion within the cytoplasm of the cell, provided the cargo is currently not traveling on the cytoskeletal network.

Every cargo starts near the surface of the nucleus within the cell model we construct (Fig. 2.1). The cargo begins diffusing (passive transport phase) from its starting position, by picking a random direction (specified by angle ϕ in Fig. 2.2) and then moving in that direction a distance a during a time interval τ (Fig. 2.2). This process continues until either the cargo reaches the cell membrane or nears a filament. Cargos are not permitted to enter the nucleus.

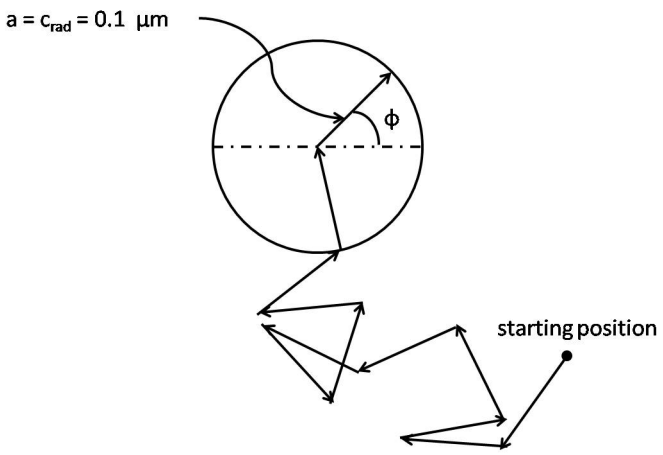

Figure 2.2: A cargo modelled as a random walker. From a starting position, the cargo chooses a random direction (given as an angle ϕ) and then moves a distance a which in our case is equal to c_{rad}, the cargo's radius. During each step, a time τ passes, which is determined by the diffusion constant.

2.2.1 Movement along a filament

When a cargo moves along the cytoskeletal network, it moves in a straight line, at a constant speed while attached to whatever filament it binds to. As the cargo approaches a filament while diffusing within the cytoplasm, if the filament is within one cargo radius of the cargo's center, the the cargo has a probability of $k_{\text{on}}\tau$ of binding to the filament (Fig. 2.3). τ is the time step in seconds and k_{on} is the rate at which cargos bind to filaments, per second. If the cargo binds to the filament, it will then move along the filament in the (+) direction (the direction of polarization) at a constant speed.

The distance the cargo moves in one timestep is then:

$$a = v\tau \tag{2.5}$$

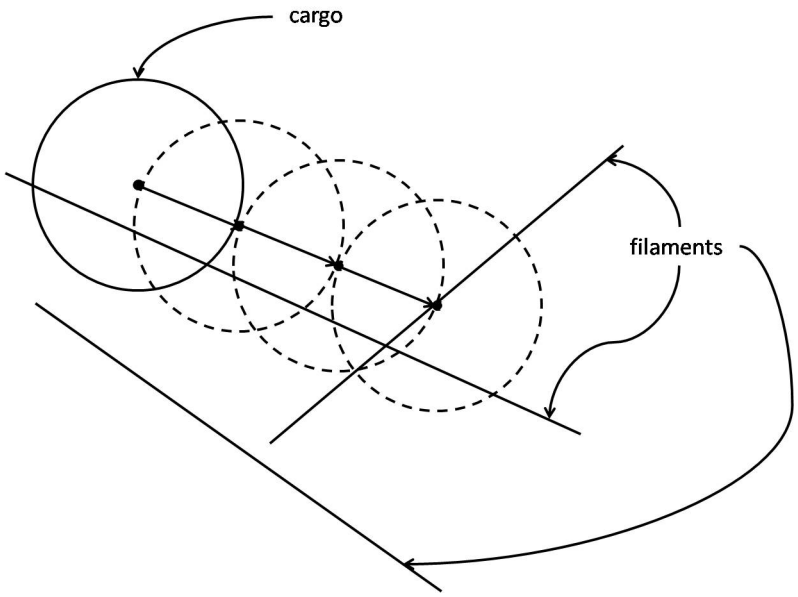

Figure 2.3: A cargo moving along a filament. If the distance between the center of the cargo and a filament is less than c_{rad}, the cargo has a chance of binding to the filament with probability $k_{\text{on}}\tau$. When the cargo is bound to the filament, every time step, it still moves a distance a. Now however, along the filament, the cargo moves in one direction with a constant speed of v. Then for every step a, a time $\tau = a/v$ passes. After each new time step, the cargo has a chance of falling off the filament with probability $k_{\text{off}}\tau$

Where v is the speed that the cargo moves along the filament. In our simulations, since the distance step, a, stays the same every time the cargo moves, the time step is now:

$$\tau = a/v \tag{2.6}$$

As the cargo moves along the filament, it has a probability of $k_{\text{off}}\tau$ of falling off. Here, τ is now the time step per length moved along the filament and k_{off} is the rate at which cargos unbinds from the filament per second (Fig. 2.3). A cargo can also detach from a filament if it walks off of its end. One thing to note here is that, in general, a cargo might switch to another filament near intersections but we are not considering that yet.

In summary, in our simulations cargos begin near the nucleus and start diffusing. Near filaments, cargos have a chance to bind to a filament and move along the direction of the filament's polarity. As the cargo moves along the filament, it has a chance of falling off during its movement and a certainty of falling off when it reaches the filament's end, after which, it will undergo diffusion again. The cargo alternates between diffusion and straight-line motion along filaments until it reaches the cell membrane. Cargos cannot pass through the nuclear membrane. When the cargo

reaches the cell membrane, its target destination, the amount of time that has passed since the cargo started moving is the first-passage time. For multiple cargos, we can get obtain the first-passage time distribution (FPTD) and the mean first-passage time (MFPT). These are quantities we can use to characterize the network. Fig. 2.4 shows some results obtained after simulating 10000 cargos. We see the FPTD as a function of time displayed. The y-axis represents the number of cargos that have escaped at each time.

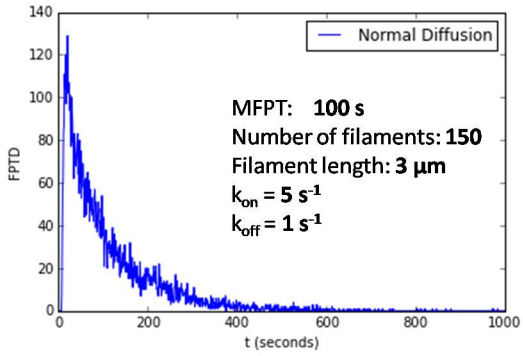

Figure 2.4: FPTD as a function of time obtained by simulating the transport of 10000 cargos over a network composed of 150 filaments with each having a length of 3 μm.

2.2.2 Random walker-Brownian motion/diffusion in detail

How far does a random walker travel after n steps?

The net displacement of the walker is [40]:

$$\vec{r}(n) = \sum_{i=1}^{n} a\hat{e}_i \tag{2.7}$$

where $\hat{e}_i$ is a unit vector pointing to the nearest-neighbor lattice site at the i^{th} step of the walk and a is the distance separation between nearest-neighbor lattice sites Since we want to quantify the behavior of multiple random-walkers, we are interested in the average, or expected value of the displacement for any particular walker:

$$\langle \vec{r}(n) \rangle = \langle \sum_{i=1}^{n} a\hat{e}_i \rangle \tag{2.8}$$

$$= a\langle \hat{e}_1 \rangle + a\langle \hat{e}_2 \rangle + \ldots + a\langle \hat{e}_n \rangle \tag{2.9}$$

$$= 0 \tag{2.10}$$

Where the last equality is due to the fact that at any particular lattice site, $\langle \hat{e}_i \rangle = 0$. This result isn't very helpful. What is more helpful is determining the mean squared displacement:

$$\langle r^2(n)\rangle = \langle(\sum_{i=1}^{n} a\hat{e}_i)^2\rangle \tag{2.11}$$

$$= \langle(a\hat{e}_1 + \ldots + a\hat{e}_n)\rangle \cdot \langle(a\hat{e}_1 + \ldots + a\hat{e}_n)\rangle \tag{2.12}$$

$$= \langle a^2 \sum^{n} (\hat{e}_i \cdot \hat{e}_i) + 2a \sum^{n} (\hat{e}_i \cdot \hat{e}_j)\rangle \tag{2.13}$$

$$= a^2 \sum^{n} \langle \hat{e}_i \cdot \hat{e}_i\rangle + 2a \sum^{n} \langle \hat{e}_i \cdot \hat{e}_j\rangle \tag{2.14}$$

$$= na^2 \tag{2.15}$$

We arrive at the last equality because $\langle \hat{e}_i \cdot \hat{e}_j\rangle = \delta_{ij}$. After n time steps, a time t has passed where $t = n\tau$ and τ is the amount of time in one time step. Then:

$$\langle r^2(t)\rangle = \frac{a^2 t}{\tau} \tag{2.16}$$

We can write this in terms of the diffusion constant, $D = a^2/(2d)\tau$

$$\langle r^2(n)\rangle = (2d)Dt \tag{2.17}$$

where d is the dimension of the lattice. The random walker undergoes Brownian motion, which is a diffusive process. Our result,

$$\langle r^2(t)\rangle \sim t \tag{2.18}$$

is a characteristic of diffusion.

2.3 Anomalous diffusion overview

The analysis of a particularly special case of diffusion, anomalous subdiffusion, is of interest to us because it is considered to be a characteristic of the passive transport phase within the cytoplasm [39]. This could possibly be the result of cargo interacting with other material within the cytoplasm and/or the binding and unbinding from the molecular motors that carry the cargo along the filaments of the cytoskeletal network [4]. Recall from before, that a characteristic of diffusion is that the mean squared displacement of the random walker as a funtion of time varies with time linearly

$$\langle r^2(t)\rangle \sim t \tag{2.19}$$

However for anomalous diffusion, the mean squared displacement varies with t as [40]:

$$\langle r^2(t)\rangle \sim t^{\alpha} \tag{2.20}$$

Anomalous diffusion with $0 < \alpha < 1$ is considered subdiffusion. We can get this result if the wait times for the random walk are taken from a power-law distribution at each time step. The model that we use is a continuous time random walk (CTRW) which is also consistent with experimental observations [41]. With our continuous-time random walk (CTRW) model we can attempt to determine the the reason for the apparent anomalous diffusive behavior of cargo observed *in vivo* within the cytoplasm. Our CTRW model is a random walk model with a constant distance step size and a time step size chosen from a distribution of waiting times:

$$\psi(t) = \begin{cases} 0 & \text{if } t < 1, \\ \alpha t^{-\alpha-1} & \text{if } t \geq 1. \end{cases} \tag{2.21}$$

Where $0 < \alpha < 1$. We can show that this distibtion of waiting times results in a mean squared displacement with a sub-linear dependence on time:

$$\langle r^2(t)\rangle \sim t^{\alpha} \tag{2.22}$$

With this result, our interest will be the effect of the subdiffusive character of the motion of cargos on the FPTD and the MFPT taken over multiple cargo simulations.

Chapter 3

Numerical Analysis

3.1 Introduction

From the perspective of the cytoskeletal network geometry, filament length, number, placement, and orientation have been shown to greatly affect transport first-passage times. For example, localizing the filament mass can optimize search and exit times [31, 33, 34], trapping regions can arise in random networks and greatly increase exit times, and orienting a small fraction of the filaments inward, towards the center of the cell can dramatically increase the mean first-passage time (MFPT) for cargo exit [31]. If trapping regions occur naturally in random network geometries, it raises the question as to how much control the cell must exert over the geometry to avoid traps or even tunably create them if cargo sequestering is desirable. In this chapter, using a numerical simulation approach, we explore how the existence of trapping regions depend on the interplay of parameters governing the cytoskeletal network geometry, in particular filament lengths and orientation.

There has been much computational work done through the use of simulations and numerical analysis in order to understand the intracellular transport process better. There are two broad classes of computational approach - (i) explicitly simulating the dynamics of a single cargo and (ii) time evolution of differential equations describing the spatial distribution of an ensemble of cargo. Explicit simulations of cargo movement typically rely on a coarse grained description of filament effects. One type of simulation, for example, involves the use of random velocity models [42] to account for ballistic transport along filaments and use this to model the spatial inhomogeneity of physical cytoskeletal networks [43]. Still, other methods focus on drawing cargo binding rates and movement information from distribution functions [44]. However, the presence of explicit filaments in models makes a qualitative difference allowing for the possibility of trapping regions, memory effects due to filament rebinding and significant changes in mean transport times [31].

A different approach is to consider the evolution of the probability distribution of a cargo ensemble. Systems of differential equations can model the time evolution of cargo spatial distributions. These tend to require the coupling of both the passive

diffusion and active transport [25] phases. A particularly simple and interesting limit of this problem occurs when filaments are aligned and motor-diven transport in the active phase facilitates advective transport in the "passive" phase [26]. In such methods, as compared to simulations of individual cargo dynamics, there is a trade-off of not requiring extensive sampling for noise reduction at the cost of precision in numerical integration upto late times. Such an approach allows for the accurate evaluation of mean first passage times but on the other hand cannot be used to evaluate stochastic variations in cargo first passage times. As is the case with most cargo dynamics simulation methods, these models do not use explicit filaments in their calculations, which produces qualitative differences as pointed out above.

Here we combine the probability distribution approach with an explicitly represented inhomogeneous cytoskeletal network whose filaments are randomly oriented in two dimensions. In our model, we will capture both the active and passive of transport through numerical integration by treating individual cargos as random walkers in the continuum limit [40] and incorporating switching between the active and passive phases by solving coupled differential equations [3] that describe the time evolution of spatial cargo distribution both on and off the explicit cytoskeletal networks. These networks are generated by placing filaments, represented by straight lines, at random locations and orientations within the cytoplasm. We will proceed in our methodology while keeping in mind the effects of filament polarization on first passage times, with the end goal of further expanding on the results of [31], which show that having a significant fraction of filaments polarized towards the center of the cell greatly slows down cargo transport.

3.2 Transition to Cargo Distributions and Elimination of Random Noise

In our work, we use the same model as that which was used to examine the cytoskeletal network topology effects on cargo first-passage times, with one major exception. We still consider the biologically relevant parameters of a circular cell with a radius of 10 μm and an inner nuclear boundary at a radius of 5 μm [31]. Within the cytoplasm, we place a randomized network of explicit cytoskeletal filaments, from which cargos can bind and unbind at rates of $k_{on} = 5s^{-1}$ and $k_{off} = 1s^{-1}$, respectively. Cargos, each with a radius of $c_{rad} = 0.1\mu m$, move while on filaments with a speed of $v = 1\mu m/s$, and move while off filaments according to a diffusion constant of $D = 0.051\mu m^2/s$. Filaments are straight lines with random locations and orientations (see [31] and Supplementary Information for more details on network generation)

Now, the discrete space that the cell occupies is a 2D lattice with lattice site locations (x, y) $0.1\mu m$ apart. The cargo distribution can exist at any (x, y) location within the cell. The filament endpoints are now placed randomly throughout the cell and the filament itself is made to be $0.2\mu m$ thick to account for the cargo radius, within which, the distribution has a chance of attaching to a filament Lattice points

within the distance of the line joining filament endpoints are treated as parts of the filament. Thus, we keep track of lattice points that are occupied by filaments.

During the diffusive transport phase, we model individual cargos as random walkers. For a distribution of cargos, we can then model its time evolution as [40]

$$\frac{\partial P(x,y,t)}{\partial t} = D\nabla^2 P(x,y,t). \tag{3.1}$$

$P(x,y,t)$ is the distribution of cargos as a function of position and time and D is the diffusion constant. Because the distribution must move via diffusion off filaments, and ballistic motion while on filaments, we can model the transport dynamics as a combination of diffusion and constant drift [3], roughly through the differential equation

$$\frac{\partial P}{\partial t} = -(\nabla \cdot \vec{v})P + D\nabla^2 P, \tag{3.2}$$

where $\vec{v}$ would be the velocity of cargos during ballistic motion.

The "on" and "off" phases of motion are distinct and well-defined. Given this, we break up (3.2) into two equations, one corresponding to an *on* distribution (P_{on}) and one corresponding to an *off* distribution (P_{off}). The active and passive phases of transport can then, respectively, be represented by

$$\frac{\partial P_{\text{on}}}{\partial t} = -(\nabla \cdot \vec{v})P_{\text{on}} \tag{3.3}$$

and

$$\frac{\partial P_{\text{off}}}{\partial t} = D\nabla^2 P_{\text{off}} \tag{3.4}$$

where, at all times, the following condition must be satisfied,

$$\frac{\partial P}{\partial t} = \frac{\partial P_{\text{on}}}{\partial t} + \frac{\partial P_{\text{off}}}{\partial t}. \tag{3.5}$$

We also have switching between these two phases of motion. For a distribution that switches between these two states [3], we can write

$$\frac{\partial P_{\text{on}}}{\partial t} = -(\nabla \cdot \vec{v})P_{\text{on}} - k_{\text{off}}P_{\text{on}} + k_{\text{on}}P_{\text{off}}, \tag{3.6}$$

$$\frac{\partial P_{\text{off}}}{\partial t} = D\nabla^2 P_{\text{off}} + k_{\text{off}}P_{\text{off}} - k_{\text{on}}P_{\text{off}}, \tag{3.7}$$

and again, a condition to be satisfied,

$$P = P_{\text{on}} + P_{\text{off}}. \tag{3.8}$$

Note that k_{on} and k_{off} are the cargo attachment and detachment rates, respectively, which couple the two differential equations.

In our numerical integrations, we find an approximate solution to (3.5) – (3.8) at each successive point in time as we let the distribution evolve. We begin implementing our integration by approximating, to first-order, the differential equations as

$$\begin{aligned}\frac{P^{n+1}_{on,i,j} - P^{n}_{on,i,j}}{\Delta t} \approx &- \frac{v_x}{2\Delta x}(P^{n}_{on,i+1,j} - P^{n}_{on,i-1,j}) - \frac{v_y}{2\Delta y}(P^{n}_{on,i,j+1} - P^{n}_{on,i,j-1}) \\ &+ (k_{on}P^{n}_{off,i,j} - k_{off}P^{n}_{on,i,j})\end{aligned} \tag{3.9}$$

and

$$\begin{aligned}\frac{P^{n+1}_{off,i,j} - P^{n}_{off,i,j}}{\Delta t} \approx &+ \frac{D}{\Delta x^2}(P^{n}_{off,i+1,j} + P^{n}_{off,i-1,j} - 2P^{n}_{off,i,j}) \\ &+ \frac{D}{\Delta y^2}(P^{n}_{off,i,j+1} + P^{n}_{off,i,j-1} - 2P^{n}_{off,i,j}) \\ &- (k_{on}P^{n}_{off,i,j} - k_{off}P^{n}_{on,i,j}).\end{aligned} \tag{3.10}$$

Here, $P^{n}_{i,j}$ is the distribution at position (i, j) in space, at time step n. $P^{n+1}_{i,j}$ will then be the distribution at the next time step $(n + 1)$. Δx and Δy are the distances between points in space and Δt is the size of the time step. v_x and v_y are the velocity components representing the speed at which the distribution moves while on a filament.

To implement our integration, we will, at each point in space (for each time step in the integration), update the probability distribution. To do this, we will first allow the distribution to either "attach" or "detach from the network. This will give us updated values for P_{off} and P_{on}:

$$\begin{aligned}P^{n+1}_{off,i,j} &= P^{n}_{off,i,j} + \Delta t \cdot (-k_{on}P^{n}_{off,i,j} + k_{off}P^{n}_{on,i,j}), \\ P^{n+1}_{on,i,j} &= P^{n}_{on,i,j} + \Delta t \cdot (k_{on}P^{n}_{off,i,j} - k_{off}P^{n}_{on,i,j}).\end{aligned} \tag{3.11}$$

We now would (roughly) allow movement off and on filaments as,

$$\begin{aligned}P^{n+1}_{off,i,j} = &P^{n+1}_{off,i,j} + \Delta t \cdot (\\ &+ \frac{D}{\Delta x^2}(P^{n}_{off,i+1,j} + P^{n}_{off,i-1,j} - 2P^{n}_{off,i,j}) \\ &+ \frac{D}{\Delta y^2}(P^{n}_{off,i,j+1} + P^{n}_{off,i,j-1} - 2P^{n}_{off,i,j})),\end{aligned} \tag{3.12}$$

and

$$\begin{aligned} P^{n+1}_{on,i,j} =& P^{n+1}_{on,i,j} + \Delta t \cdot (\\ & - \frac{v_x}{2\Delta x}(P^n_{on,i+1,j} - P^n_{on,i-1,j}) \\ & - \frac{v_y}{2\Delta y}(P^n_{on,i,j+1} - P^n_{on,i,j-1})). \end{aligned} \tag{3.13}$$

The first equation, (3.12), is fine written just as it is. The implementation of movement on the network is not as simple as it is presented in (3.13), however. As it's written, there is the implication that the "on" distribution can move anywhere throughout the cell. This is not the case though, as P_{on} can only be nonzero where a filament exists at (i, j). We take this into account and also assume that the distribution "walks" off the ends of filaments. Refer to the Supplementary Information in order to see our refined, complete versions of P_{off} and P_{on}.

After properly updating P_{off} and P_{on}, we can calculate the total probability distribution at each point in space for each time step n,

$$\begin{aligned} P^n_{off,i,j} &= P^{n+1}_{off,i,j} \\ P^n_{on,i,j} &= P^{n+1}_{on,i,j} \\ P^n_{i,j} &= P^n_{off,i,j} + P^n_{on,i,j}. \end{aligned} \tag{3.14}$$

If we integrate through enough time steps, we can obtain first-passage time information. At every instance in time, we can determine the probability that the distribution has stayed within the cell, the survival probability, $S(t)$ by integrating P over its domain (the interior of the cell). The rate that the survival probability changes in time gives us the first-passage time distribution (FPTD) ($F(t)$ below):

$$S(t) = \int_{\text{domain}} P(x, y, t)dxdy, \tag{3.15}$$

$$F(t) = -\frac{\partial S(t)}{\partial t}. \tag{3.16}$$

By averaging over the FPTD, we get the MFPT,

$$\text{MFPT} = \int_0^\infty tF(t)dt. \tag{3.17}$$

In practice, we only integrate to the time where the probability distribution is no longer leaving the cell. We can compare the FPTDs and MFPTs obtained through integration with those which are obtained through the simulation of the transport of multiple cargos. Fig. 3.1 provides a comparison.

Fig. 3.1a shows a FPTD obtained through the simulation of the movement of 10000 cargos. It can clearly be seen that there is some noise inherent in the simulation itself. This is different from what can be seen in Fig. 3.1b, where the FPTD shown

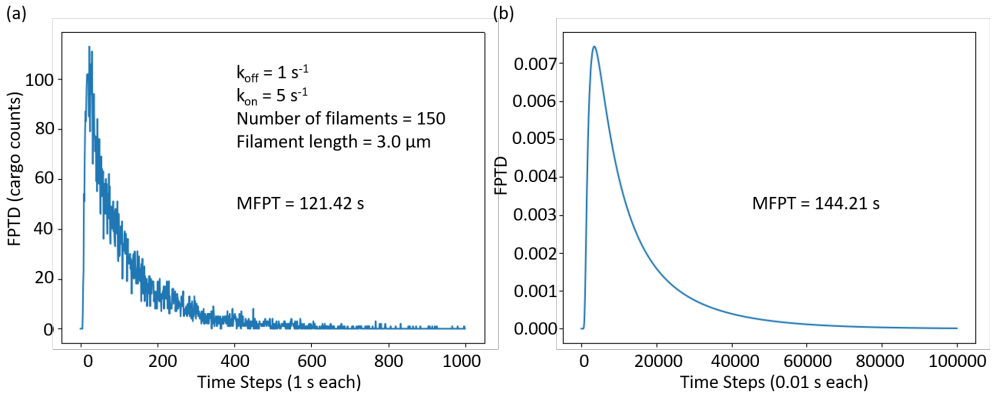

Figure 3.1: A comparison of FPTD achieved via (a) simulation of 10000 cargos and (b) numerical integration. Notice that the MFPTs are comparable and that the FPTD in (b) is smoother.

was obtained through an integration as outlined above. All noise is gone from the FPTD, which indicates the advantage of accuracy of the numerical integration method over the simulation method.

3.3 Distribution Evolution on Networks of Different Polarization Biases

Before completing the calculations of (3.15), (3.16), and (3.17), we examine the state of the distribution at an intermediate time step, for networks of different polarization biases to get a sense of the distribution evolution process. We define the network polarization bias as the probability that each filament in the network has of being polarized outward, towards the cell membrabe, away from the nucleus. For example, a network with a polarization bias of 0.1 will have 10% of its filaments polarized toward the cell membrane (i.e., having a polarization of +1), and 90% of its filaments polarized towards the nucleus (having a polarization of -1). Fig. 3.2 shows four different networks with four different polarization biases. Each network containes 150 filaments, each with length of 5 μm.

It is apparent in Fig. 3.2a, that all filaments (colored red) have a polarization of -1. Thus, the polarization bias for the network is 0.0. In Fig. 3.2b, some of the filaments have a polarization of +1 (they are colored blue). As the network is generated randomly, and each filament is placed down, each has a 10% chance of obtaining a polarization of +1. This is why the polarization bias for this network is 0.1. The networks in Figs. 3.2c and 3.2d are generated similarly, with Fig. 3.2c showing a network with approximately 70% of its filaments having a polarization of +1, and Fig. 3.2d showing a network with 100% of its filaments having a polarization of +1.

Similar to how the positions of cargos are initialized in [31], the cargo distribution

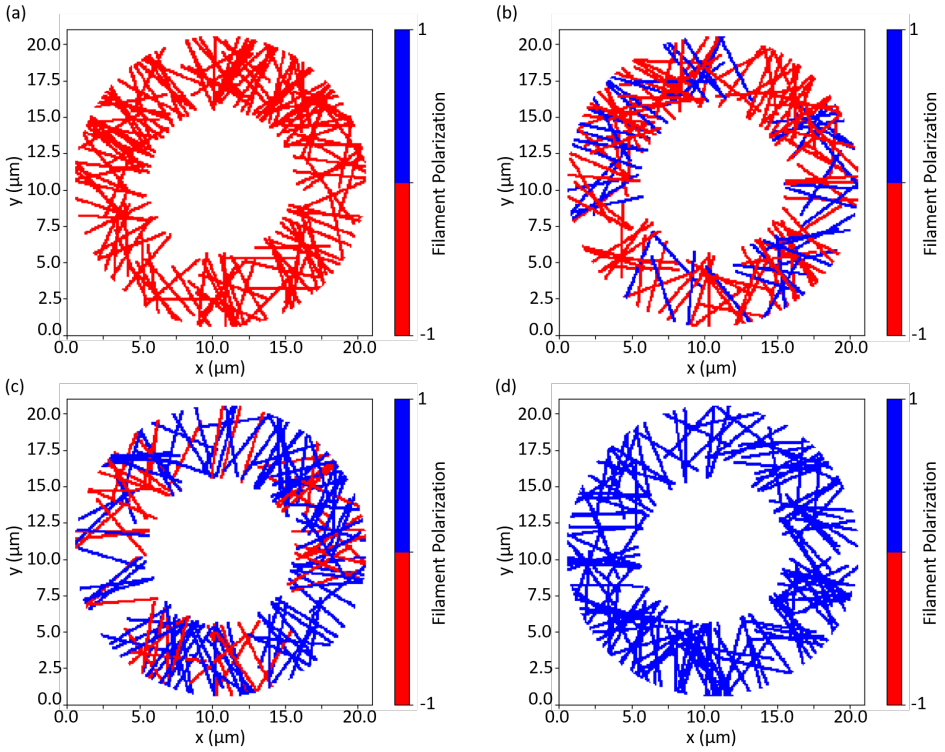

Figure 3.2: Different polarization biases for 150 filaments, each with a length of 5 μm. The polarization biases are (a) 0.0 (0 % of filaments pointing outward), (b) 0.3 (approximately 30 % of filaments pointing outward), (c) 0.7, and (d) 1.0.

begins as an annulus of width 0.2 μm near the surface of the nucleus, off all filaments. We then allow the distribution to evolve in time. In Fig 3.3 we see the state of the distribution after it is able to evolve over four different networks for 100 s. The distribution evolves over the networks shown in Fig. 3.2.

Note, as one would expect, that an increasing polarization bias enhances the distribution's ability to reach the cell membrane. Especially, consider the case of Fig. 3.3a, where the polarization bias is 0.0. Much of the distribution is still near the nucleus, which likely indicates that the MFPT for cargo distribution on this network will be very high.

3.4 Survival Probability Analysis for Different Filament Lengths and Polarizations

After sufficient lengths of time, the distribution is able to leave the cell. To determine the effect that filament length and network polarization bias has on the ability of the distribution to leave the cell, we allow the the initial probability disribution to evolve for 1000 s over networks of different combinations of filament lenghts and

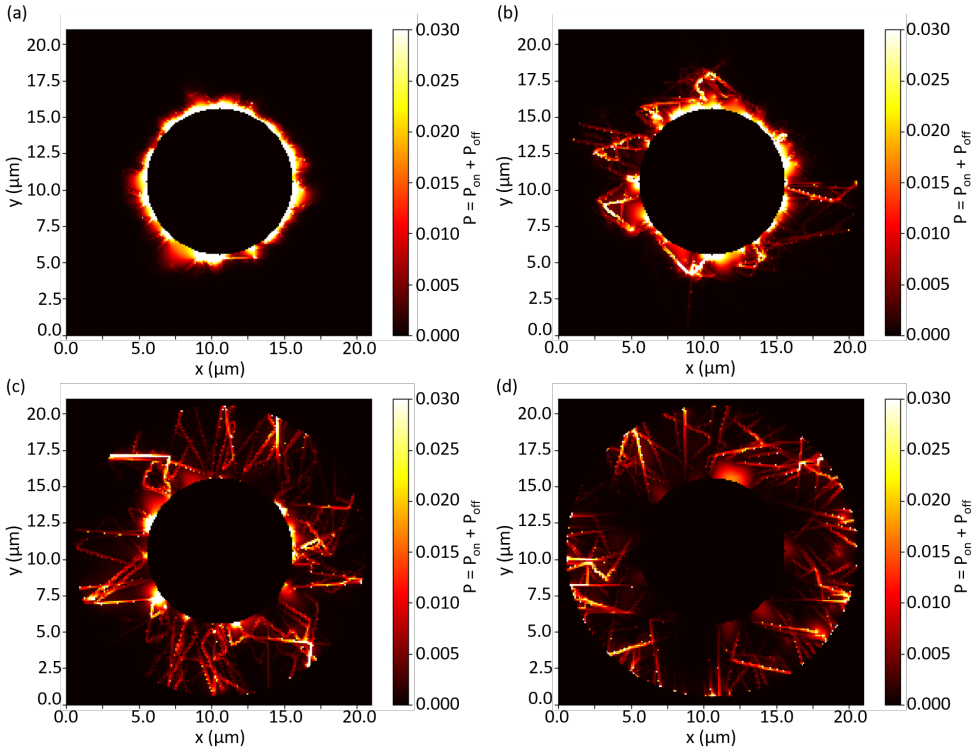

Figure 3.3: State of the cargo distribution after moving (for 100 s) on and off a cytoskeletal network comprised of 150, 5 μm filaments for polarization biases of (a) 0.0, (b) 0.3, (c) 0.7, and (d) 1.0. The distribution is evolved over the networks in Fig. 3.2

polarization biases. We keep the number of filaments constant at 150. Before doing any first-passage time analysis, it is important to determine that a sufficient amount of the distribution has left the cell (i.e., the survival probabiilty is low and the FPTD is near zero), by the time the integration is stopped, in order for any calculated MFPT to have any meaning. We can see in Fig. 3.4a that for filament lenghts of 1, 2, 3, 4, and 5 μm, and for network polarization biases of 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6, the survival probability is mostly nonzero and even approaches 1.0 for low polarization biases.

Beyond a polarization bias of 0.7 and for all filament lengths, the survival probability is 0.0, meaning that MFPT calculations have obvious meaning. The MFPT is plotted in Fig. 3.4b for these polarization biases. The results indicate what one would expect. As the filament length and/or the polarization bias for the network increases, the MFPTs decrease in value. Because MFPT calculations only have meaning at sufficiently high polarization biases, we choose to analyze the survival probability for different network realizations at low polarzation biases. One thing to note in Fig. 3.4a that we will come back to in the analyses to come, is the clear transition from high to low survival probability as the polarization bias is increased while the filament

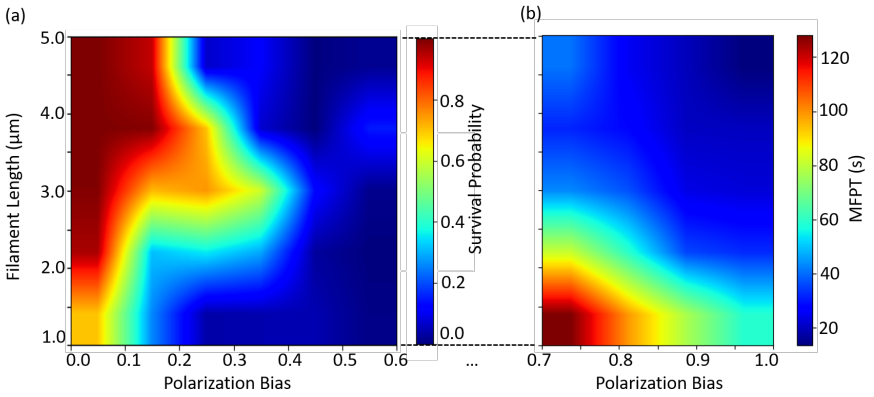

Figure 3.4: (a) Survival probability at 1000 s as a function of filament length and polarization bias. At a polarization bias of approximately 0.7, the entire distribution has left the cell at this point in time. (b) The MFPT for different filament lengths and polarization biases in the regime where the survival probability is zero. As one would expect, MFPT decreases with increasing filament length and polarization bias

length is held constant.

3.5 Intermediate Filament Lengths Enhance Survival Probability

In order to make sure that the polarization bias effect that we referred to in the previous paragraph is not just a network-dependent effect, we calculate the survival probability for the probability distrubution at 1000 s on five different networks at each filament length, polarization bias combination (filament lengths of 1, 2, 3, 4, and 5 μm, and polarization biases of 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0) and average the results. We can see in Fig. 3.5a that the transition of the survival probability from high to low values is still present, and it is in fact delayed at intermediate filament lengths (particularly at a filament length of 3 μm).

Near this transition, it can be seen in Fig. 3.5b that the network to network survival probability standard deviation possesses its highest values. To further demonstrate this point, we make a 2D plot displaying the average survival probability as a function of network polarization bias for different filament lengths. Note in particular that at a filament length of 3 μm and a polarization bias of 0.2, the survival probability is greater than for any other filament length at this polarization bias, and that the standard deviation (given by the size of the error bars) is relatively large as well.

These results are consistent with the findings of [31], where, as filament polarizations were changed from +1 to -1 when the filaments were 3 μm in length, significant increases in MFPT were found. Here, we witness this transition in the form of the survival probability which, as we can see from (3.36), (3.37), and (3.38), is directly related to the MFPT.

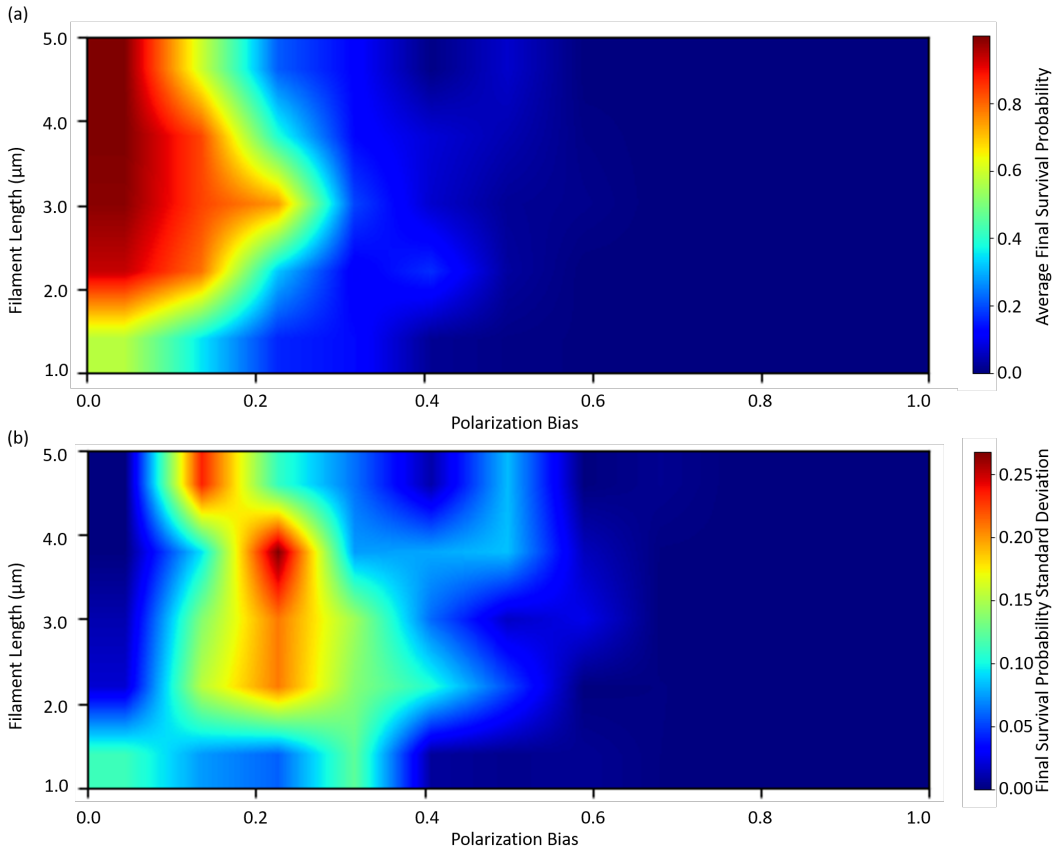

Figure 3.5: The survival probability was calculated for five different networks at each filament length, polarization bias. (a) Shows the average survival probability for different filament lengths and polarization biases. For low polarization biases, the survival probability decreases sharply at all filament lengths. At intermediate filament lengths, the survival probability maintains a relatively high value at higher polarization biases than it does for shorter and longer filaments. (b) The network to network standard deviation of the survival probability. It can be seen that the variance is highest near the transition from high to low survival probability.

3.6 Intermediate Filament Lenghts Facilitate Distribution Trapping in the Bulk

To further explore the effects seen in the previous section, we evaluate the behavior of the probability distribution moving across single networks at nine different points in the filament length and polarization bias phase space. The points in phase space at which we will be examing the probability distribution more closely are indicated by the white dots in Figs. 3.7a and 3.7b. These nine points are the (polarization bias, filament length) values of (0.1, 5.0 μm), (0.3, 5.0 μm), (0.5, 5.0 μm), (0.1, 3.0

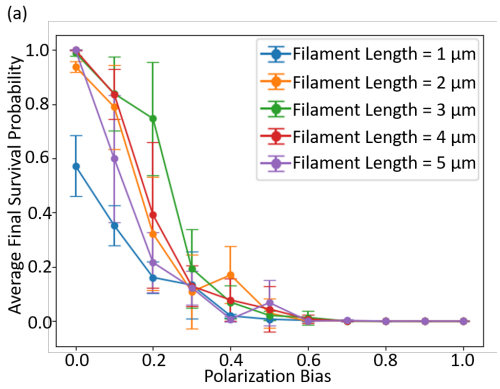

Figure 3.6: The average survival probability, as in Fig. 3.5a, but for different filament lengths, as a function of polarization bias. The error bars are the standard deviations calculated for Fig. 3.5b.

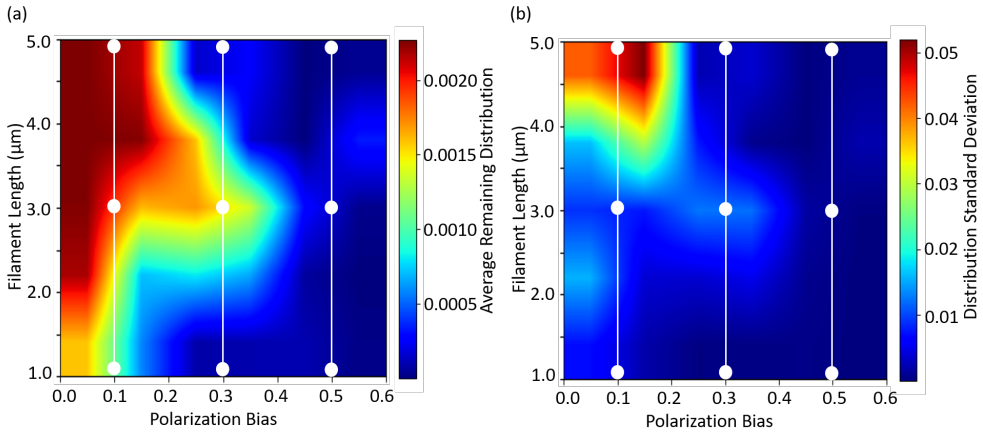

Figure 3.7: (a) The survival probability for different filament lengths and polarization biases for one network per length-bias combination. (b) The location-based standard deviation of the remaining probability distribution after 1000 s. We will be able to see the significance of these two figures by examining the remaining distribution for nine different filament lengths and polarization biases (indicated by the white dots). It will be particularly useful to compare the distributions at the same polarization bias, but different filament lenghts (see the white lines connecting the dots).

μm), (0.3, 3.0 μm), (0.5, 3.0 μm), (0.1, 1.0 μm), (0.3, 1.0 μm), and (0.5, 1.0 μm).

In Fig. 3.7, we plot the survival probability (Fig. 3.7a) next to the location-based standard deviation of the probability distribution after evolving in time for 1000 s (Fig. 3.7b). We see that the distribution standard deviation is highest when the filament length is high and the polarization bias is low. In order to understand why this is, we must examine the actual distribution at this point in time more closely.

Fig. 3.8 shows the state of the probability distribution after evolving in time for 1000 s at the nine points in phase space. Most of our analysis can be done by

paying attention to the distribution as it moves across networks containing filaments of lengths 5 μm and 3 μm. In Fig. 3.8a, where the polarization bias is 0.1 and the filament length is 5 μm, much of the distrubution is still near the nucleus. When the filaments are 5 μm long, the distribution can only be in a "trapped" state near the nucleus. If any part of the distribution makes it to the middle of the bulk, it will likely either be directed out the cell, or right back to the nucleus. The latter situation is more likely to happen when the polarization bias is low.

The radial position of the distribution in Fig. 3.8a is in contrast to where the distribution appears in Fig. 3.8c. In this situation, the filament lengths are 3 μm even though the polarization bias is still 0.1. Here, the distribution can be seen gathering at bright spots in the figure even near the middle of the bulk. At these filament lengths, trapping regions can occur more uniformly throughout the cell. This helps to offer an explanation to what can be seen in Fig. 3.7, where although the survival probability is relatively high at a polarization bias of 0.1 for filament lengths of 5 μm and 3 μm, the distribution standard deviation is much lower when the filament length is 3 μm.

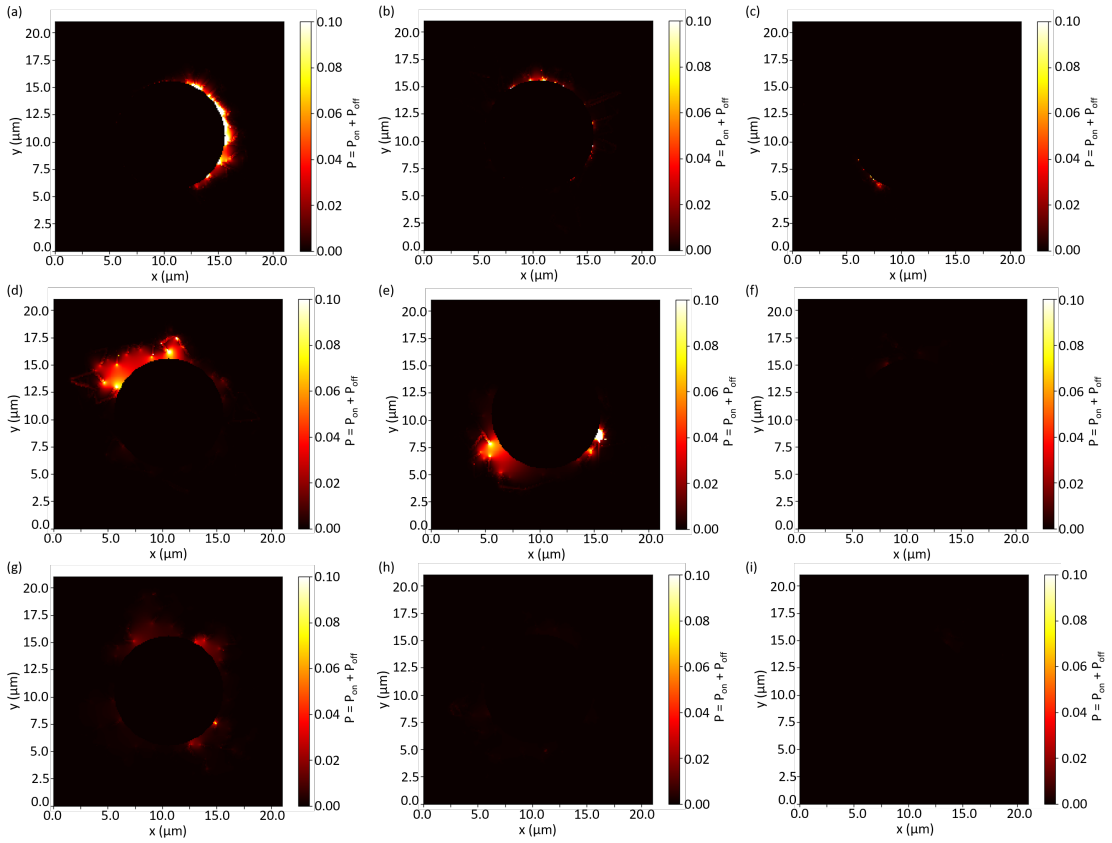

Figure 3.8: The remaining cargo distribution after 1000 s when the filament length is 5 μm and the polariztion biases are (a) 0.1, (b) 0.3, and (c) 0.5, the filament length is 3μm and the polarization biases are (d) 0.1, (e) 0.3, and (f) 0.5, and when the filament length is 1 μm and the polarization biases are (g) 0.1, (h) 0.3, and (i) 0.5. (a) shows that much of the distribution is still contained in traps near the nucleus, as this is the only area traps can occur when the filament length is 5.0 μm. Compare this with (d), where the filament length is 3.0 μm and distribution traps are more spread out within the bulk cytoplasm. This helps explain the results seen in Fig. 3.7, where at a polarization bias of 0.1, the survival probability is relatively high at filament lengths of 5.0 μm and 3.0 μm (shown in Fig. 3.7a), but the standard deviation is much lower at a filament length of 3.0 μm than it is at 5.0 μm, as can be seen by examining Fig. 3.7b. In comparing (b), (e), and (h), it can be seen that greatest amount of distribution is remaining at a filament length of 3.0 μm when the polarization bias is 0.3. This reflects the larger survival probability at intermediate filament lengths when the polarization bias is this high. In looking at (c), (f), and (i), it can be seen that most of the distribution has left the cell by this time.

The results depicted in Figs. 3.8b and 3.8e reflect the delayed transition of the survival probability from high to low values at intermediate filament lengths. In Fig. 3.8b, we can see that most of the distribution has left the cell and whatever remains is still near the nucleus. In Fig. 3.8e, much of the distribution is still in the cell,

sometimes gathered in regions indicative of the presence of filaments smaller than 5 μm. These figures make even more sense if they are compared to Fig. 3.7a. In following the center white line in this figure (drawn at a polarization bias of 0.3), it can be seen, when starting at the top of the plot, the survival probability starts low (when the filament length is 5 μm), then increases in value (when the filament length is decreased to 3 μm), and then drops in value again (at a filament length of 1 μm). These results show that having a network comprised of filaments of intermediate lengths helps to facilitate both the maintenance of a high survival probability, and a large probability distribution standard deviation.

3.7 Conclusion and Future Directions

We have shown that it is possible to create a model that involves integrating a probability distribution of cargos, with respect to time, as the distribution moves both on and off an explicit cytoskeletal network. The ability to numerically integrate a distribution of cargos, rather than just having to rely on individual cargo simulations, allows for the extraction of more accurate cargo survival probability, and thus, first-passage time information. This is most apparent when comparing FPTDs obtained through the simulation of cargo transport with those obtained through integrating distributions of cargos.

What we were able to show with this model is the sensitivity of distribution survival probability to the lengths of the filaments that make up the cytoskeletal network, as well as the network polarization bias. The most interesting results are seen when the filament lengths are near 3 μm and the polarization bias of the network is near 0.3. With this combination of network parameters, a high survival probability is maintained and the distribution can be in a so-called trapped state nearly uniformly throughout the cell. Although we were able to achieve these results with this model, it still has room to expand. For example, it is possible to set up particular cytoskeletal network geometries rather than just placing filaments randomly. We can model the network to more closely follow the more realistic biological setup where microtubules are oriented radially outward and a relatively thin layer of actin filaments lies near the cell membrane. This provides us the opportunity to compare to simulations that have made use of this geometry [42].

One thing we have neglected to take into account is the effect of multiple-filament intersections on the transport of cargos. These intesections can cause increased molecular motor-based tug-of-war when multiple motors are present on a single cargo [17,45] as well as the formation of cargo vortices and cycling behavior [27]. These effects, along with the fact the the interior of the cell is actually dense creating crowding effects [46], imply that normal diffusion is not necessarily a sufficient explanation for the passive transport phase. A way to incorporate anomalous diffusion into the model may, therefore, be needed.

Also complicating matters are the effects that the filament endpoints have on cargo binding and unbinding rates. There is even the issue of whether or not the cargos will

walk off at all when they reach filament endpoints, which may actually depend on the motor type and the filament polymerization rate [47]. Additionally, the possibilty of cargo crowding near endpoints may even help facilitate molecular motor dissociation, meaning that as cargos approach filament ends, they unbind higher rates [48]. These physical findings mean that our model has room to be refined and improved upon in the future.

3.8 Supplementary Information

3.8.1 Continuum Limit - Numerical integration of a distribution of cargos

3.8.2 Diffusion

We examine the random walker in the continuum limit to determine the behavior of the initial distribution of walkers [40]. By integrating these distributions over time, we can get more accurate answers for the FPTD and the MFPT than we do in our simulations because we are able to eliminate random noise. The behavior of a distribution of random walkers as a function of time obeys the following differential equation

$$\frac{\partial P(x,y,t)}{\partial t} = D\nabla^2 P(x,y,t) \tag{3.18}$$

Where D is again the diffusion constant. In this situation, D is constant and diffusion is normal. We derive (3.18) as follows.

3.8.3 Random walker-Brownian motion/diffusion: Continuum limit

We examine the random walker in the continuum limit to determine the behavior of a distribution of walkers. Consider a walker on a two-dimensional lattice. A walker at lattice site (i, j), arriving at time step $n + 1$ came from one of four places, with equal probability:

$$P_{n+1}(i,j) = \frac{1}{4}(P_n(i-1,j) + P_n(i+1,j) + P_n(i,j-1) + P_n(i,j+1)) \tag{3.19}$$

$P_{n+1}(i, j)$ is the probability that a walker exists at (i, j) at time step $n + 1$. The equality comes from the fact that probability is conserved. If we consider the lattice on an xy plane, again with nearest-neighbor lattice separation being a,

$$P_{n+1}(x,y) = \frac{1}{4}(P_n(x-a,y) + P_n(x+a,y) + P_n(x,y-a) + P_n(x,y+a)) \tag{3.20}$$

If we subtract $P_n(x, y)$ from both sides, we can make some approximations

$$P_{n+1}(x,y) - P_n(x,y) = \frac{1}{4}P_n(x-a,y) + \frac{1}{4}P_n(x+a,y) \tag{3.21}$$

$$+\frac{1}{4}P_n(x,y-a) + \frac{1}{4}P_n(x,y+a) \tag{3.22}$$

$$-P_n(x,y) \tag{3.23}$$

Again, assume that at the nth time step, time t has passed. Each time step occurs over a duration of time τ. Now, notice that, to first-order,

$$\frac{\partial P(x,y,t)}{\partial t} \approx \frac{P(x,y,t+\tau) - P(x,y,t)}{\tau} \tag{3.24}$$

We also have the approximations:

$$\frac{\partial^2 P(x,y,t)}{\partial x^2} \approx \frac{P(x+a,y,t) + P(x-a,y,t) - 2P(x,y,t)}{a^2} \tag{3.25}$$

$$\frac{\partial^2 P(x,y,t)}{\partial y^2} \approx \frac{P(x,y+a,t) + P(x,y-a,t) - 2P(x,y,t)}{a^2} \tag{3.26}$$

After some algebra, we then have:

$$\frac{\partial P(x,y,t)}{\partial t} = \frac{a^2}{4\tau}\left(\frac{\partial^2 P(x,y,t)}{\partial x^2} + \frac{\partial^2 P(x,y,t)}{\partial y^2}\right) \tag{3.27}$$

Notice that $a^2/4\tau$ is just the diffusion constant, D in two dimensions. This gives us the result:

$$\frac{\partial P(x,y,t)}{\partial t} = D\nabla^2 P(x,y,t) \tag{3.28}$$

This is the diffusion equation for a distribution of random walkers.

3.8.4 Movement along a filament

For the distribution of cargos within the cell, which can involve transport both on and off filaments, we can model the dynamics as a combination of diffusion and constant drift [3].

$$\frac{\partial P}{\partial t} = -(\nabla \cdot \vec{v})P + D\nabla^2 P \tag{3.29}$$

Because there are are two distinct types of motion (motion on and motion off of the filaments), we can break up the differential equation here into two, one correspondng to an *on* and one corresponding to an *off* distribution.

The motion of the phases of passive and active transport can then be represented by:

$$\frac{\partial P_{\text{on}}}{\partial t} = -(\nabla \cdot \vec{v})P_{\text{on}} \tag{3.30}$$

$$\frac{\partial P_{\text{off}}}{\partial t} = D\nabla^2 P_{\text{off}} \tag{3.31}$$

Where

$$\frac{\partial P}{\partial t} = \frac{\partial P_{\text{on}}}{\partial t} + \frac{\partial P_{\text{off}}}{\partial t} \tag{3.32}$$

3.8.5 Bringing it all together

We also have switching between these two phases of motion. For a distribution that switches between these two states [3],

$$\frac{\partial P_{\text{on}}}{\partial t} = -(\nabla \cdot \vec{v})P_{\text{on}} - k_{\text{off}}P_{\text{on}} + k_{\text{on}}P_{\text{off}} \tag{3.33}$$

$$\frac{\partial P_{\text{off}}}{\partial t} = D\nabla^2 P_{\text{off}} + k_{\text{off}}P_{\text{off}} - k_{\text{on}}P_{\text{off}} \tag{3.34}$$

Where, at all times,

$$P = P_{\text{on}} + P_{\text{off}} \tag{3.35}$$

We integrate these two differential equations to get the FPTD and the MFPT for our distribution of cargos that begins near the surface of the nucleus.

At every instance in time, we can determine the probability that the distribution has stayed within the cell, the survival probability, $S(t)$ by integrating P over its domain (the interior of the cell). The rate that the survival probability changes in time gives us the FPTD ($F(t)$ below):

$$S(t) = \int_{\text{domain}} P(x, y, t)dxdy, \tag{3.36}$$

$$F(t) = -\frac{\partial S(t)}{\partial t}. \tag{3.37}$$

By averaging over the FPTD, we get the MFPT,

$$\text{MFPT} = \int_0^\infty tF(t)dt. \tag{3.38}$$

3.8.6 Selecting an appropriate timestep: von Neumann Stability Analysis

We wish to select a timestep for approximating the solutions to the differential equations such that the resulting probability distribution does not diverge. To achieve this, our time step must be sufficiently small. We can determine an appropriate timestep by performing a "rough" von Neumann Stability Analysis for our system of equations.

The equations are

$$\frac{\partial P_{\text{on}}}{\partial t} = -(\nabla \cdot \vec{v})P_{\text{on}} - k_{\text{off}}P_{\text{on}} + k_{\text{on}}P_{\text{off}}, \tag{3.39}$$

$$\frac{\partial P_{\text{off}}}{\partial t} = D\nabla^2 P_{\text{off}} + k_{\text{off}}P_{\text{off}} - k_{\text{on}}P_{\text{off}} \tag{3.40}$$

where, at all times,

$$P = P_{\text{on}} + P_{\text{off}}. \tag{3.41}$$

To first order, the above differential equations (in 2D) approximate to

$$\begin{aligned}\frac{P^{n+1}_{on,i,j} - P^{n}_{on,i,j}}{\Delta t} \approx &- \frac{v_x}{2\Delta x}(P^{n}_{on,i+1,j} - P^{n}_{on,i-1,j}) - \frac{v_y}{2\Delta y}(P^{n}_{on,i,j+1} - P^{n}_{on,i,j-1}) \\ &+ (k_{on}P^{n}_{off,i,j} - k_{off}P^{n}_{on,i,j})\end{aligned} \tag{3.42}$$

and

$$\begin{aligned}\frac{P^{n+1}_{off,i,j} - P^{n}_{off,i,j}}{\Delta t} \approx &+ \frac{D}{\Delta x^2}(P^{n}_{off,i+1,j} + P^{n}_{off,i-1,j} - 2P^{n}_{off,i,j}) \\ &+ \frac{D}{\Delta y^2}(P^{n}_{off,i,j+1} + P^{n}_{off,i,j-1} - 2P^{n}_{off,i,j}) \\ &- (k_{on}P^{n}_{off,i,j} - k_{off}P^{n}_{on,i,j}).\end{aligned} \tag{3.43}$$

In our computations, we wish to select a Δt so that iterations of P^n do not diverge. We can begin this process by assuming solutions of the form

$$P^{n}_{on,l,m} = \tilde{P}^{n}_{on} e^{i(k_x l\Delta x + k_y m\Delta y)} \tag{3.44}$$

and

$$P^{n}_{off,l,m} = \tilde{P}^{n}_{off} e^{i(k_x l\Delta x + k_y m\Delta y)}. \tag{3.45}$$

Then, after some algebra and substituting these solutions into the equations above, as well as making use of the fact that

$$cos(x) = \frac{e^{ix} + e^{-ix}}{2}, \tag{3.46}$$

$$sin(x) = \frac{e^{ix} - e^{-ix}}{2i}, \tag{3.47}$$

and

$$sin^2(x) = \frac{1 - cos(2x)}{2}, \tag{3.48}$$

we get

$$\begin{aligned}\tilde{P}^{n+1}_{on} = &- \frac{iv_x\Delta t\tilde{P}^{n}_{on}}{2\Delta x}sin(k_x\Delta x) - \frac{iv_y\Delta t\tilde{P}^{n}_{on}}{2\Delta y}sin(k_y\Delta y) \\ &+ (k_{on}\tilde{P}^{n}_{off} - k_{off}\tilde{P}^{n}_{on}) + \tilde{P}^{n}_{on}\end{aligned} \tag{3.49}$$

and

$$\begin{aligned}\tilde{P}_{off}^{n+1} =& -\frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta x^2}sin^2(\frac{k_x\Delta x}{2}) - \frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta y^2}sin^2(\frac{k_y\Delta y}{2}) \\ & - (k_{on}\tilde{P}_{off}^{n} - k_{off}\tilde{P}_{on}^{n}) + \tilde{P}_{off}^{n}. \end{aligned} \tag{3.50}$$

We add these two equations together to get

$$\begin{aligned}\tilde{P}^{n+1} =& -\frac{iv_x\Delta t\tilde{P}_{on}^{n}}{2\Delta x}sin(k_x\Delta x) - \frac{iv_y\Delta t\tilde{P}_{on}^{n}}{2\Delta y}sin(k_y\Delta y) \\ & - \frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta x^2}sin^2(\frac{k_x\Delta x}{2}) - \frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta y^2}sin^2(\frac{k_y\Delta y}{2}) \\ & + \tilde{P}^{n}. \end{aligned} \tag{3.51}$$

After subtracting $\tilde{P}^n$ from both sides, then dividing it and rearranging terms we get

$$\begin{aligned}1 - \frac{\tilde{P}^{n+1}}{\tilde{P}^{n}} =& + \frac{iv_x\Delta t\tilde{P}_{on}^{n}}{2\Delta x\tilde{P}^{n}}sin(k_x\Delta x) + \frac{iv_y\Delta t\tilde{P}_{on}^{n}}{2\Delta y\tilde{P}^{n}}sin(k_y\Delta y) \\ & + \frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta x^2\tilde{P}^{n}}sin^2(\frac{k_x\Delta x}{2}) + \frac{4D\Delta t\tilde{P}_{off}^{n}}{\Delta y^2\tilde{P}^{n}}sin^2(\frac{k_y\Delta y}{2}). \end{aligned} \tag{3.52}$$

To ensure that each forward iteration of $\tilde{P}^n$ does not lead to divergence, we must always have

$$1 - |\frac{\tilde{P}^{n+1}}{\tilde{P}^{n}}| < 1. \tag{3.53}$$

We can be sure of this if we notice that the right side of (3.52) is a complex number of the form

$$z = a + ib, \tag{3.54}$$

$$|z| = \sqrt{a^2 + b^2}, \tag{3.55}$$

and then have

$$|z| = \sqrt{a^2 + b^2} < 1. \tag{3.56}$$

To enforce this, we choose to have

$$a, b < \frac{\sqrt{2}}{2}. \tag{3.57}$$

Now, let

$$a = \frac{4D\Delta t\tilde{P}^n_{off}}{\Delta x^2\tilde{P}^n} sin^2(\frac{k_x\Delta x}{2}) + \frac{4D\Delta t\tilde{P}^n_{off}}{\Delta y^2\tilde{P}^n} sin^2(\frac{k_y\Delta y}{2}) < \frac{\sqrt{2}}{2}. \tag{3.58}$$

Given that

$$max(\frac{\tilde{P}^n_{off}}{\tilde{P}^n} sin^2\theta) = 1, \tag{3.59}$$

we must select a Δt such that

$$\frac{4D\Delta t}{\Delta x^2} + \frac{4D\Delta t}{\Delta y^2} < \frac{\sqrt{2}}{2}. \tag{3.60}$$

Solving for Δt we get

$$\Delta t < \frac{\Delta x^2\Delta y^2\sqrt{2}}{8D(\Delta x^2 + \Delta y^2)}. \tag{3.61}$$

Now for b,

$$b = \frac{v_x\Delta t\tilde{P}^n_{on}}{2\Delta x\tilde{P}^n} sin(k_x\Delta x) + \frac{v_y\Delta t\tilde{P}^n_{on}}{2\Delta y\tilde{P}^n} sin(k_y\Delta y) < \frac{\sqrt{2}}{2}. \tag{3.62}$$

Similar to the case with a,

$$max(\frac{\tilde{P}^n_{on}}{\tilde{P}^n} sin\theta) = 1, \tag{3.63}$$

which means we must have a Δt such that

$$\frac{v_x\Delta t}{2\Delta x} + \frac{v_y\Delta t}{2\Delta y} < \frac{\sqrt{2}}{2}. \tag{3.64}$$

In other terms,

$$\Delta t < \frac{\Delta x\Delta y\sqrt{2}}{2(v_y\Delta x + v_x\Delta y)}. \tag{3.65}$$

We can then be sure that the Δt we use in our computations is small enough if we select the minimum of (3.61) and (3.65):

$$\Delta t_{chosen} = min(\frac{\Delta x^2\Delta y^2\sqrt{2}}{8D(\Delta x^2 + \Delta y^2)}, \frac{\Delta x\Delta y\sqrt{2}}{2(v_y\Delta x + v_x\Delta y)}) \tag{3.66}$$

In our system, in keeping consistent with the parameter values chosen for our simulations, we have $\Delta x = \Delta y = 0.1\mu m$, $D = 0.051\mu m^2/s$, and $max(v_x) = max(v_y) = 1\mu m/s$. We then have.

$$\Delta t_{chosen} = min(0.017s, 0.035s) \tag{3.67}$$

In our numerical integration, we end up choosing $\Delta t_{chosen} = 0.01s$. With this selection, we do in fact get that the probability distribution is conserved after each time iteration and that it does not diverge, meaning that Δt_{chosen} is small enough.

Now that we have all of the parameter values we need, we can determine how to carry out the integration.

3.8.7 Integrating the differential equations in our system

We can use the first-order approximations given by (3.42) and (3.43) to get us started in determining an adequate integration methodology. What we will do, is at each point in space (i, j) (for each time step in the integration, n), update the probability distribution. To do this, we will first allow the distribution to either "attach" or "detach from the network. This will give us updated values for P_{off} and P_{on}:

$$\begin{aligned} P^{n+1}_{off,i,j} &= P^n_{off,i,j} + \Delta t \cdot (-k_{on} P^n_{off,i,j} + k_{off} P^n_{on,i,j}), \\ P^{n+1}_{on,i,j} &= P^n_{on,i,j} + \Delta t \cdot (k_{on} P^n_{off,i,j} - k_{off} P^n_{on,i,j}). \end{aligned} \tag{3.68}$$

We now would (roughly) allow movement off and on filaments as,

$$\begin{aligned} P^{n+1}_{off,i,j} =& P^{n+1}_{off,i,j} + \Delta t \cdot (\\ &+ \frac{D}{\Delta x^2}(P^n_{off,i+1,j} + P^n_{off,i-1,j} - 2P^n_{off,i,j}) \\ &+ \frac{D}{\Delta y^2}(P^n_{off,i,j+1} + P^n_{off,i,j-1} - 2P^n_{off,i,j})), \end{aligned} \tag{3.69}$$

and

$$\begin{aligned} P^{n+1}_{on,i,j} =& P^{n+1}_{on,i,j} + \Delta t \cdot (\\ &- \frac{v_x}{2\Delta x}(P^n_{on,i+1,j} - P^n_{on,i-1,j}) \\ &- \frac{v_y}{2\Delta y}(P^n_{on,i,j+1} - P^n_{on,i,j-1})). \end{aligned} \tag{3.70}$$

The first equation, (3.69), is fine written just as it is. The implementation of movement on the network, is not as simple as it is presented in (3.70), however. As it's written, there is the implication that the "on" distribution can move anywhere throughout the cell. This is not the case though, as P_{on} can only be nonzero where a filament exists at (i, j). To take this into account, let $\delta_{fil,i,j} = 1$ where a filament exists. At these positions, $v_{x,i,j}$ and $v_{y,i,j}$ will also be nonzero (corresponding to cargo x and y velocity components along filaments). (3.70) then becomes,

$$
\begin{aligned}
P^{n+1}_{on,i,j} =& P^{n+1}_{on,i,j} + \Delta t \cdot \delta_{cell,i,j} \cdot (\\
& (\frac{1}{2}(v_{x,i,j} + |v_{x,i,j}|)) \cdot (-\frac{P^n_{on,i,j}}{\Delta x}) \cdot (\delta_{cell,i+1,j}) \cdot (\delta_{fil,i+1,j}) \\
& + (\frac{1}{2}(v_{x,i-1,j} + |v_{x,i-1,j}|)) \cdot (\frac{P^n_{on,i-1,j}}{\Delta x}) \cdot (\delta_{cell,i,j}) \cdot (\delta_{fil,i,j}) \\
& + (\frac{1}{2}(v_{x,i,j} - |v_{x,i,j}|)) \cdot (\frac{P^n_{on,i,j}}{\Delta x}) \cdot (\delta_{cell,i-1,j}) \cdot (\delta_{fil,i-1,j}) \\
& + (\frac{1}{2}(v_{x,i+1,j} - |v_{x,i+1,j}|)) \cdot (-\frac{P^n_{on,i+1,j}}{\Delta x}) \cdot (\delta_{cell,i,j}) \cdot (\delta_{fil,i,j}) \\
& + (\frac{1}{2}(v_{y,i,j} + |v_{y,i,j}|)) \cdot (-\frac{P^n_{on,i,j}}{\Delta y}) \cdot (\delta_{cell,i,j+1}) \cdot (\delta_{fil,i,j+1}) \\
& + (\frac{1}{2}(v_{y,i,j-1} + |v_{y,i,j-1}|)) \cdot (\frac{P^n_{on,i,j-1}}{\Delta y}) \cdot (\delta_{cell,i,j}) \cdot (\delta_{fil,i,j}) \\
& + (\frac{1}{2}(v_{y,i,j} - |v_{y,i,j}|)) \cdot (\frac{P^n_{on,i,j}}{\Delta y}) \cdot (\delta_{cell,i,j-1}) \cdot (\delta_{fil,i,j-1}) \\
& + (\frac{1}{2}(v_{y,i,j+1} - |v_{y,i,j+1}|)) \cdot (-\frac{P^n_{on,i,j+1}}{\Delta y}) \cdot (\delta_{cell,i,j}) \cdot (\delta_{fil,i,j})), \qquad (3.71)
\end{aligned}
$$

where $\delta_{cell,i,j} = 1$ outside the reflecting inner boundary of the cell. Missing in (3.71) is what happens at filament endpoints. Similar to what we did in our simulations, we assume cargos walk off the end of filements. This action causes both P_{off} and P_{on} to change. We define a filament endpoint as the end that cargos walk towards. We will have $\delta_{end,i,j} = 1$ where a filament end exists. Given this, P_{off} and P_{on} are updated even further through

$$
\begin{aligned}
P^{n+1}_{off,i,j} =& P^{n+1}_{off,i,j} + \Delta t \cdot \delta_{cell,i,j} \cdot (\\
& (\frac{1}{2}(v_{x,i-1,j} + |v_{x,i-1,j}|)) \cdot (\frac{P^n_{on,i-1,j}}{\Delta x}) \cdot (\delta_{end,i-1,j}) \\
& + (\frac{1}{2}(v_{x,i+1,j} - |v_{x,i+1,j}|)) \cdot (-\frac{P^n_{on,i+1,j}}{\Delta x}) \cdot (\delta_{end,i+1,j}) \\
& + (\frac{1}{2}(v_{y,i,j-1} + |v_{y,i,j-1}|)) \cdot (\frac{P^n_{on,i,j-1}}{\Delta y}) \cdot (\delta_{end,i,j-1}) \\
& + (\frac{1}{2}(v_{y,i,j+1} - |v_{y,i,j+1}|)) \cdot (-\frac{P^n_{on,i,j+1}}{\Delta y}) \cdot (\delta_{end,i,j+1})) \qquad (3.72)
\end{aligned}
$$

and

$$
\begin{aligned}
P^{n+1}_{on,i,j} =& P^{n+1}_{on,i,j} + \Delta t \cdot \delta_{cell,i,j} \cdot (\\
& (\frac{1}{2}(v_{x,i,j} + |v_{x,i,j}|)) \cdot (-\frac{P^n_{on,i,j}}{\Delta x}) \cdot (\delta_{cell,i+1,j}) \cdot (\delta_{end,i,j}) \\
& + (\frac{1}{2}(v_{x,i,j} - |v_{x,i,j}|)) \cdot (\frac{P^n_{on,i,j}}{\Delta x}) \cdot (\delta_{cell,i-1,j}) \cdot (\delta_{end,i,j}) \\
& + (\frac{1}{2}(v_{y,i,j} + |v_{y,i,j}|)) \cdot (-\frac{P^n_{on,i,j}}{\Delta y}) \cdot (\delta_{cell,i,j+1}) \cdot (\delta_{end,i,j}) \\
& + (\frac{1}{2}(v_{y,i,j} - |v_{y,i,j}|)) \cdot (\frac{P^n_{on,i,j}}{\Delta y}) \cdot (\delta_{cell,i,j-1}) \cdot (\delta_{end,i,j})). \qquad (3.73)
\end{aligned}
$$

If we apply, in the following order, (3.68), (3.69), (3.71), (3.72), and (3.73) to the probability distribution, we are able to update it in one time step. Now, we can calculate the total probability distribution at each point in space for each time step,

$$
\begin{aligned}
& P^n_{off,i,j} = P^{n+1}_{off,i,j} \\
& P^n_{on,i,j} = P^{n+1}_{on,i,j} \\
& P^n_{i,j} = P^n_{off,i,j} + P^n_{on,i,j}. \qquad (3.74)
\end{aligned}
$$

With the total probability calculated we can apply (3.36), (3.37), and (3.38) to calculate the survival probability, the FPTD and the MFPT, respectively.

Chapter 4

Anomalous Diffusion

4.1 Introduction

Although we have established the importance of network architecture in transport, it is worth noting that the overall transport process is dependent on diffusion in the passive phase as well. While previous studies [31] assumed that the passive diffusive was characterized by normal Brownian diffusion, in the context of the crowded cytoplasm [46], diffusion is known to be anomalous [39]. Experiments involving measurements of diffusion of cargo after filament depolymerization in both extracts [39] and in cells [4] have shown anomalously subdiffusive behavior. In fact, anomalous diffusion can be used to describe the entire intracellular transport process. The active transport phase is super diffusive while anomalous sub-diffusion is considered to be a characteristic of the passive transport phase within the bulk cytoplasm [3].

In this chapter, we explore how the interplay between super-diffusive transport, provided by explicitly modeled cytoskeletal filaments, and the anomalous nature of sub-diffusion in the bulk, can lead to novel effects in transport behavior at the cellular scale. In particular, we are interested in how the geometric properties of the cytoskeletal network dictated by the lengths and density of the constituent filaments influence transport in the presence of anomalous sub-diffusive transport in the bulk cytoplasm, and especially whether they can be tuned to access different transport phases. Anomalous diffusion can generally be described by two prevailing models, Fractional Brownian Motion (FBM) [49], which is an ergodic process and Continuous Time Random Walk (CTRW) [50], which is not. In the context of intracellular transport, CTRW has been shown to describe bulk diffusion when filaments are shortened *in vivo* [4,39,41] as well as in the presence of cargo interactions with filaments [41] and vortices and cycling behavior near actin filament intersections in the case of multiple molecular motors [27]. It has also been observed that diffusive cytosolic transport is best explained by a CTRW, while filament transport is best represented by FBM [39]. While different mechanisms have been proposed in these papers, their relative contributions to the observed CTRW behavior is not clear yet and is beyond the scope of this manuscript. The goal of our paper is to show how the observed CTRW for

passive cargo diffusion in conjunction with active transport on cytoskeleton structures influence the overall transport properties. Since we use explicit filament networks, we only need to account for anomalous sub-diffusion in the bulk in our model, which we therefore do, using CTRW.

Given our focus on understanding the basic physics of the interplay between superdiffusive network transport subdiffusive cytoplasmic transport, we choose to only consider the simplest geometries for the cytoplasmic boundaries and cytoskeletal networks. Our model, introduced in section 2, consists of a circular cell with a concentric circular nucleus and a randomly oriented filament network between the nuclear and cellular membranes (see Fig.1). In this case, the geometric properties of the cytoskeletal network are dictated by the lengths and density of the constituent filaments. We simulate the transport of cargos, starting at the nucleus, in the center of the cell, and alternating between ballistic transport along the filaments and sub-diffusive transport in the bulk, till they reach their target destination - the outer cell membrane. For the sake of simplicity, and, in order to focus only on relevant parameters such as filament length, concentration and dwell time statistics of anomalous diffusion, we neglect the elasticity of the filaments [51], viscoelastic interactions between cargos, motors, and the network [52], confinement effects [29, 30] and scenarios involving cargos carried by multiple motors [16] (we consider only single-motor active transport). As the simulation unfolds, we measure mean squared displacements (MSDs) as a function of time and the distributions of first-passage times (FPTDs) to get to the destination for cargos over multiple filament networks for varying network parameters (filament length and concentration). Because we explicitly model the filament geometry, we are also able to compute the variance in these measurements across multiple network realizations with the same parameters. We should emphasize here that we focus on quantities like the MSD, the time averaged MSD and first passage time distributions because they give us physiologically relevant information like overall transit times and also because they are readily and typically measured quantities in microscopy experiments. Therefore this approach allows us to reveal the signatures of underlying anomalous processes in macroscopic and averaged observables that are readily experimentally accessible.

To begin with, we consider the case of pure cytosplasmic subdiffusion in the absence of filaments. We verify that our implementation of CTRW produces the desired behavior for both ensemble averaged and time averaged MSDs. We then add filaments to the system. We show, over a physiologically relevant range of filament lengths and numbers, that the network introduces a superdiffusive phase at early times which crosses over to a phase where the CTRW is dominant and produces subdiffusion at late times. We also show that the superdiffusive phase is most sensitive to filament length. Finally, we apply our simulation approach to the problem of insulin secretion from pancreatic cells, which is characterized, in healthy cells, by a quick release of a large fraction of granules followed by a low but sustained rate of release at late times after glucose stimulation [53]. We show that the superdiffusive phase introduced by the filament network manifests as a peak in the secretion at early times followed

by an extended sustained release phase that is dominated by the CTRW process at late times. Our results are consistent with *in vivo* observations of insulin transport and shed light on the potential for the cell to tune transport phases by altering its cytoskeletal network.

4.2 Methods

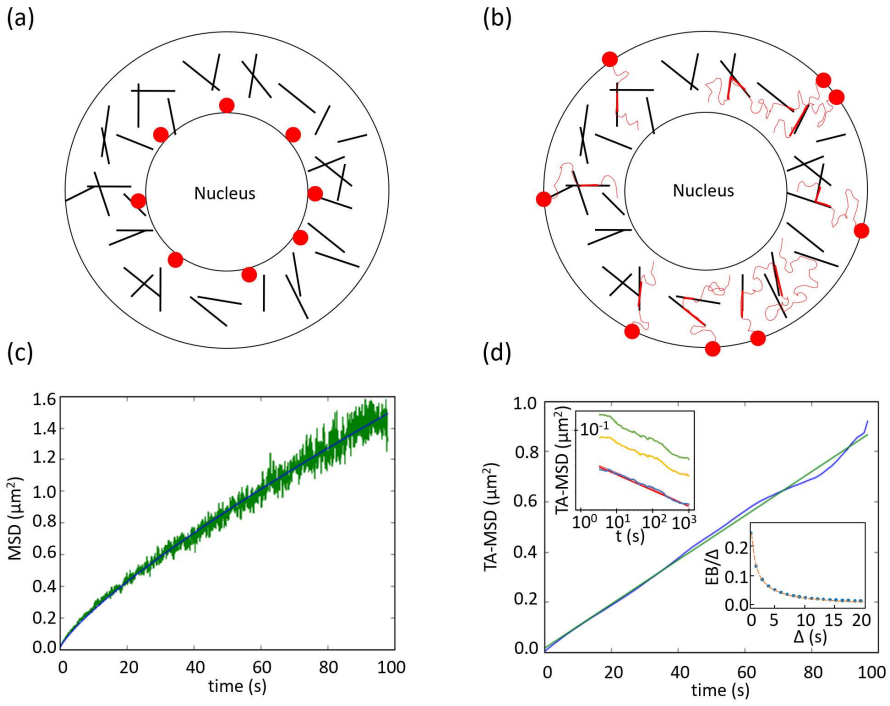

Figure 4.1: (a) The initial state of the system. Cargos (red symbols) start near the nucleus. Randomly placed filaments (black lines) model the cytoskeleton. Each filament has a fixed polarization. (b) The final state of the simulation. Cargos alternate between passive and active phases of transport until they reach the outer cell membrane. Individual trajectories are denoted by light red curves. (c) The ensemble-average MSD for a system of 1000 cargos, with no filaments present (CTRW only)-green curve. The dark blue line is a power-law fit with an exponent $\alpha = 0.8$. (d) TA-MSD for the same system for a constant measuring time, t, as a function of a sliding time window Δ. Inset, upper-left shows the TA-MSD for constant time windows, as a function of measuring time. Inset, lower-right shows the Ergodicity-Breaking parameter plotted as EB/Δ as a function of Δ (dashed line shows $1/\Delta$).

We build on previous work [31] in which simulations of cargos alternate between phases of ballistic motion along filaments (corresponding to active transport) and

random walk phases resulting in Brownian motion/normal diffusion in the bulk (corresponding to passive transport). For our simulations, we consider a model eukaryotic cell consisting of a nucleus, cell membrane and filaments that make up the cytoskeleton. We use biologically realistic parameters for the various processes involved [31] (see Supplementary Information Table 1) and implement all of our simulations in 2D, in order to better compare our results with experiments, where processes are typically observed in a 2D plane. The cell, then, is represented by a 2D disk with a radius of 10 μm, while the nucleus has a radius of 5 μm. Filaments are straight lines with random locations and orientations (see [31] and Supplementary Information for more details on network generation). Cargos have a radius of 100 nm and bind to filaments with a rate of $k_{on} = 5s^{-1}$, and unbind from filaments at a rate $k_{off} = 1s^{-1}$. The cargo radius only influences the diffusion constant and the range of interaction of cargos. Cargos begin near the nucleus (Fig. 4.6a) and undergo transport until they reach the cell membrane (Fig. 4.6b) while alternating between phases on and off the filament network. Off the network, the diffusion constant (in the case of normal diffusion) is $D = 0.051\mu m^2/\text{s}$ and while traveling on the network, cargos move at a speed of $v = 1\mu m/s$.

In this work, we extend the previous model [31] by accounting for the fact that cargos can undergo anomalous subdiffusion instead of regular diffusion during the passive phase. A signature of anomalous diffusion is that the cargos have a mean squared displacement (MSD) that scales as

$$\langle r^2(t)\rangle \sim t^{\alpha} \tag{4.1}$$

with $0 < \alpha < 1$ indicating subdiffusion. In order to incorporate anomalous diffusion in our simulations, we have cargos perform a CTRW during the passive transport phase. To implement this, we select a waiting or dwell time between successive random walk steps, from the distribution

$$\psi(t) = \begin{cases} 0 & \text{if } t < 1, \\ \alpha t^{-\alpha-1} & \text{if } t \geq 1. \end{cases} \tag{4.2}$$

with $0 < \alpha < 1$, which we will show leads to the anomalous diffusion signature of (4.1) in the supplementary section of this chapter. After waiting for the selected time, the cargo moves a distance of 0.1 μm, with the maximum cargo movement speed being set by the diffusion constant. Experiments with cargo in cell extracts [39] have shown that, in the presence of microtubules, cargos move with a measured α of about 1.4-1.5, but when the filaments are depolymerized, α values between 0.65 and 0.98 were observed. These results seem to indicate that diffusion in the absence of any filaments, due to the bulk alone, is subdiffusive with an exponent of about 0.8. This value is also consistent with the subdiffusive exponent observed for insulin granules in pancreatic cells that had been treated by vinblastine to depolymerize filaments [4]. Based on these and other [41] similar results, we use $\alpha = 0.8$ in most of our simulations, unless otherwise specified.

4.3 Validating MSD scaling and aging due to CTRW

We begin our simulations with a test of our system in the absence of any filaments. Here, cargos begin near the nucleus and undergo purely passive transport (CTRW only) until they reach the outer membrane. For purely CTRW transport with a distribution of wait times defined by Eq. 4.2, we expect the MSD to scale according to Eq. 4.1. Fig. 4.6c shows the ensemble averaged MSD from our simulations, which agrees very well with the expected power law scaling with an exponent of 0.8.

Since CTRW is a non-ergodic process, we also analyze time-average mean squared displacement (TA-MSD) data. By definition, this value is given by [54]:

$$\overline{\delta^2}(\Delta, t) = \frac{\int_0^{t-\Delta}[x(t'+\Delta) - x(t')]^2 dt'}{t-\Delta} \tag{4.3}$$

where Δ is the sliding time window (time between measurements) and t is the total measuring time. In the limit where $\Delta << t$, averaging over many cargos yields

$$\langle\overline{\delta^2}\rangle \sim \frac{\Delta}{t^{1-\alpha}} \tag{4.4}$$

From Fig. 4.6d (main), we see that the measured TA-MSD increases linearly with Δ, as expected. In Fig. 4.6d (upper-left inset), we plot scaling of the TA-MSD from simulations with measuring time t for different values of $\Delta = 1s, 2s, 3s$. We again recover the expected scaling behavior, $t^{1-\alpha}$. Finally, we also plot the measured ergodicity breaking (EB) parameter,

$$EB = \frac{\langle(\overline{\delta^2})^2\rangle - \langle\overline{\delta^2}\rangle^2}{\langle\overline{\delta^2}\rangle^2} \tag{4.5}$$

in 4.6d (lower-right inset) as EB/Δ, which scales as $\sim 1/\Delta$ as expected for CTRW [4], signifying convergence of EB to a nonzero constant value, another characteristic feature of CTRW. Taken together, these results indicate that our CTRW model implementation is effective in producing anomalous subdiffusion with the desired exponent.

4.4 Adding filaments introduces a superdiffusive phase

Having validated and created a baseline for the MSD scaling in the subdiffusive passive phase, we now consider the addition of filaments, creating a cytoskeletal network. We add to the network 100, 200, 300, 400, and 500 filaments, with lengths of 1, 2, 3, 4, and 5 μm (details of network generation in [31] and Supplementary Information; the range of filament numbers and lengths are consistent with reasonable in vivo values [31]). The most notable difference is observed in the ensemble-average MSD. We can see in Fig. 4.2a that, in contrast to the case with no filaments present (data in blue), the

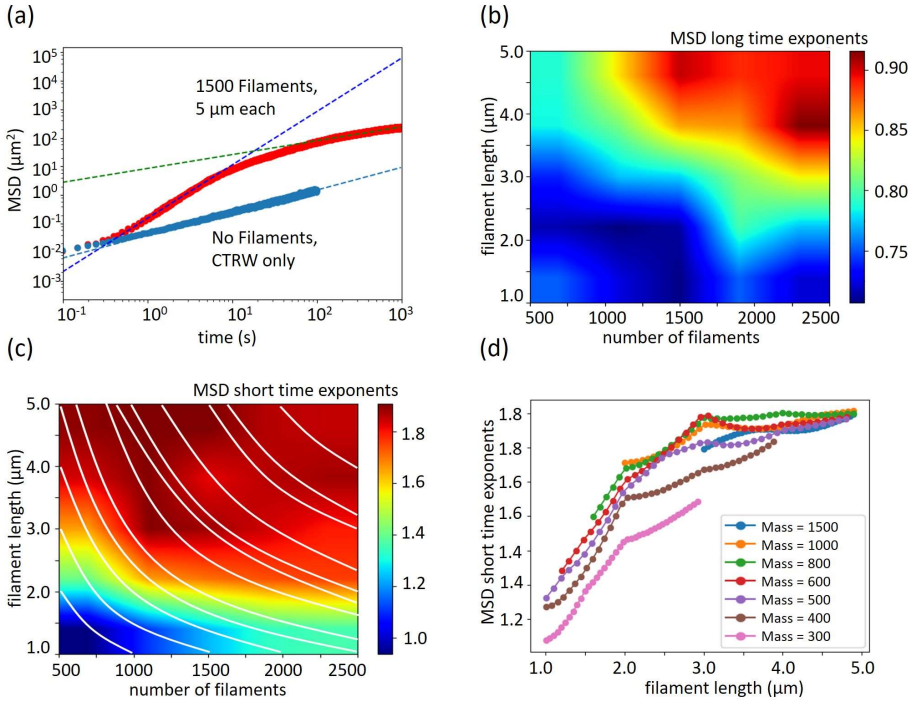

Figure 4.2: (a) A log-log plot of an ensemble-average MSD in the presence of filaments (red data, 1500 filaments, 5 μm each) compared to MSD for CTRW only (below, blue data). Dashed lines show fits to different power law behaviors for short and long times for the MSD data with filaments and over the entire time range for the control CTRW only case. The measured long (b) and short-time (c) power-law exponents as a function of filament length and number. In (c), lines of constant mass are in white. (d) MSD short-time exponents as a function of filament length for different total filament masses. Averaging is over N=10000 cargo in all cases. Error in the measured exponents due to fitting is less than 6% over the parameter range explored.

MSD in the presence of filaments (data shown in red) shows different scaling behaviors in different time regimes. Fitting the MSD in the two time regimes, we can see that the short-time slope (dashed blue line) is larger than 1 (indicating superdiffusion with an MSD scaling exponent larger than 1) and is distinctly larger than the long-time slope (dashed green line) which is below 1 (indicating subdiffusion). Thus at early times, it appears that the MSD is dominated by movement along the filaments, giving rise to superdiffusion. At later times, past some transition time set by the typical timescale for which a cargo walks on a filament before detaching (between 1s and 10s), we can see a crossover to CTRW dominated behavior, as suggested by comparing the slope of this second regime with the slope of the CTRW only data. To understand how these different exponents depend on the network parameters, we plot the MSD scaling

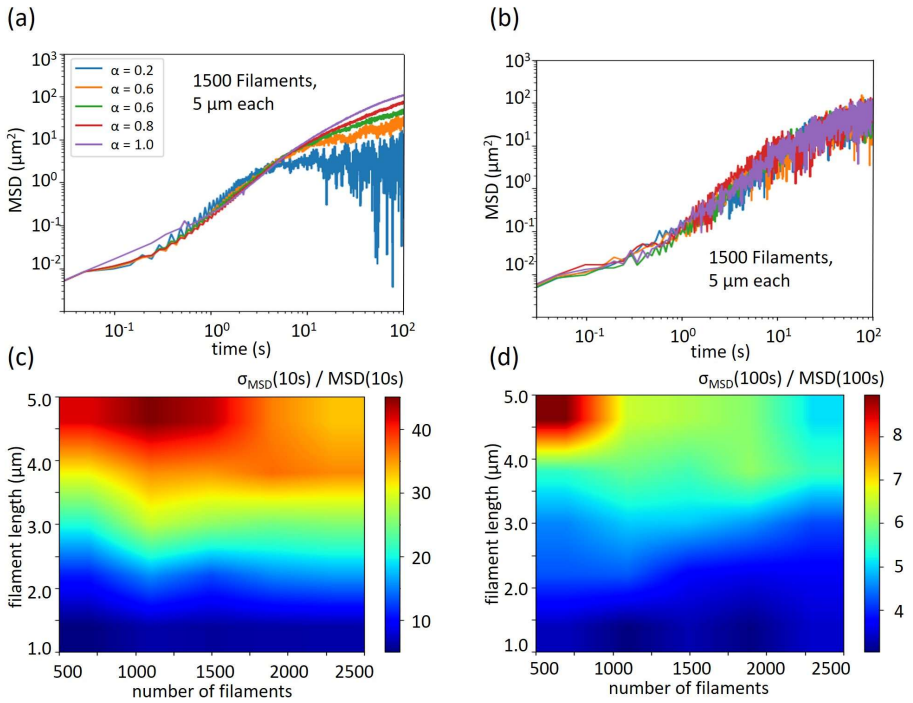

Figure 4.3: (a) Ensemble-average MSD as a function of time for different values of α. (N=100 cargo) (b) MSD as a function of time for 100 cargos over 5 different networks at fixed $\alpha = 0.8$. Normalized standard deviation of MSD (averaged over 100 cargo and 100 different networks) at 10 s (c) and 100 s (d) as a function of filament length and number.

exponents in the long-time (Fig. 4.2b) and the short-time (Fig. 4.2c) regimes as a function of number and lengths of the filaments. Consistent with the picture that the long time dynamics are controlled by CTRW, the long time exponents are all close to 0.8 and fairly insensitive to filament density and number, except at the very highest network masses, where the signature of the short time superdiffusive phase begins to show. Note that the exponent appears to go below 0.8 at low densities because of confinement effects from the boundary and, as expected, this effect diminishes with increasing cell radius (see Supplementary Information). Not surprisingly, the network parameters have the greatest effect on the MSD at shorter times, where the slope is greatest. The short time exponent changes all the way from 1 (or diffusive) at the lowest network masses to almost ballistic (~ 1.8) at high network masses. To examine the relative importance of filament length and density, we consider curves of constant mass (white lines in Fig. 4.2c), where filament mass is defined as the number of filaments multiplied by the length of each filament. In Fig. 4.2d, we plot the short-time MSD exponent as a function of filament length for different network

masses (corresponding to the lines in Fig. 4.2c). We see from the rough collapse of the curves that the short-time exponent shows very modest increases with greater mass at fixed filament length but is much more sensitive to the filament length for constant mass. This indicates that it is the filament length, not the total mass of the filaments, that is an important factor in driving the MSD at short times.

We now look more closely at the long time behavior to understand how it is controlled by the CTRW. Fig. 4.3a plots the MSD for values of α from 0.2 to 1 in the presence of 1500 filaments of length 5 μm. We can see the effect of the dwell time distribution on the MSD in the long-time regime (Fig. 4.3a). Whereas the MSD is controlled by the filament network at early times and is insensitive to α, decreasing α leads to a decrease in the MSD at late times. Because we are interested in how the geometry of the network itself affects MSD, we next consider how the MSD varies across different network realizations. Fig. 4.3b plots the MSD for five different networks, each with 300 filaments of length, 5 μm. We immediately see that any difference between them is within the intrinsic variance on each network due to the CTRW, suggesting that the variance due to the dwell time distribution dominates over network geometry effects. To quantify this further, we simulate the transport of 100 cargos over 100 networks and calculate the MSD at 10s and 100s and track its variance at those times. Figs. 4.3c and 4.3d show the standard deviations of the MSD at 10s and 100s (normalized by the mean MSD at those times), respectively, for different filament lengths and numbers. The normalized standard deviation increases with increasing filament length and decreasing numbers of filaments, with the effect being much more pronounced at early times when the network geometry is influential.

4.5 Tuning transport phases using network parameters

Of particular interest due to its relevance to real biological processes such a secretion and exocytosis is the time taken to transport cargo to the peripheral cell membrane. We can quantify this transport by measuring the time that it takes for cargo to first reach the outer membrane and constructing a first passage time distribution (FPTD) from these times. Such FPTDs can have distinctive features that arise from the underlying transport processes. For example, it has been shown that insulin secretion in healthy pancreatic cells, where insulin containing vesicles are transported to the membrane and secreted outside of the cell [4], is characterized by a distinctly "biphasic" FPTD, consisting of an initial spike, followed by a long, sustained release of insulin [53]. In a recent model [4] used to explain this process, insulin granules move throughout the cell through a combination of FBM and CTRW until they reach some distance a from a fast-releasing hot spot on the cell membrane, where the particles move only via FBM [53, 55]. As the parameter a is increased, there is an initial peak of insulin flux followed by a more stable phase, giving the biphasic behavior seen in experimental observations. While the distance a is meant to model a region with no

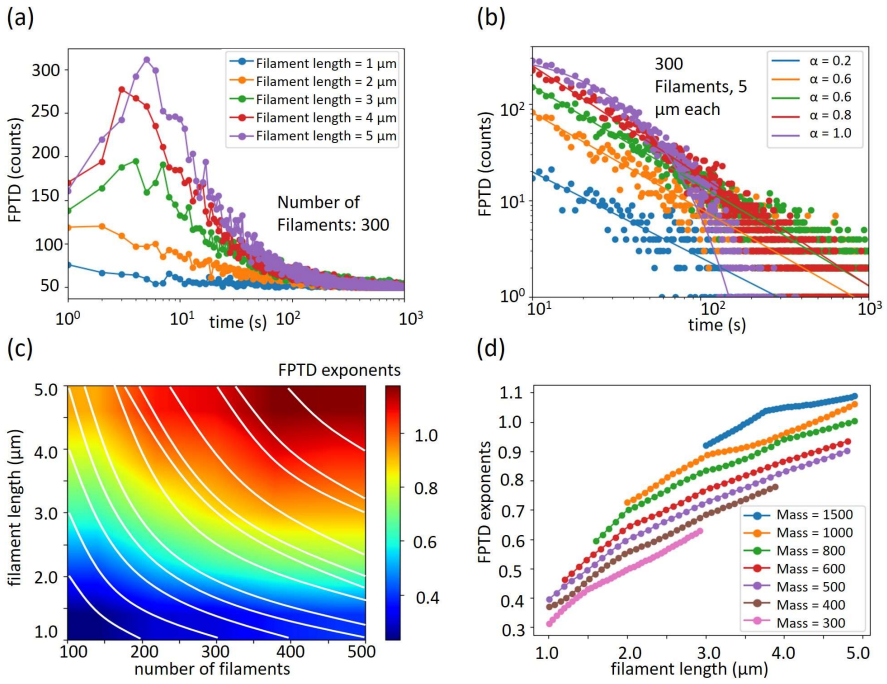

Figure 4.4: (a) FPTDs for networks comprised of 300 filaments, with varying lengths. (b) The second phase of the FPTD for different values of α. (c) Strength of FPTD decay as a function of filament length and number. Lines of constant mass are in white. (d) FPTD decay exponent as a function of filament length for different filament masses. FPTDs are for 100 cargo over 100 different networks. Error in measured exponents due to fitting is less than 6% over the parameter range explored.

trapping, it is not clear what the physical cytoskeletal architecture would be corresponding to this parameter. While the insulin secretion process as a whole is complex involving many signals, regulatory proteins, fusion proteins and motor proteins such as myosins and kinesins [53], *in vivo* observations suggest that the cytoskeletal network has an important part to play in this process and, in particular, that depolymerization and rearrangement of actin filaments seen during glucose stimulation is one of the key regulators [53, 56, 57]. Here, we consider the network filaments explicitly and are therefore able to directly examine the result of filament depolymerization in isolation. The exact features of the secretion profile depend on the parameters and also assumptions about the initial distribution of insulin granules. Rather than trying to replicate that, we focus on two main features observed in the biphasic secretion - fast secretion upon stimulation and sustained slow secretion, at later times. We use pure CTRW to represent anomalous diffusion in the bulk and, instead of the parameter a, we vary, as in the case of our MSD analysis, explicit filament length and number. We

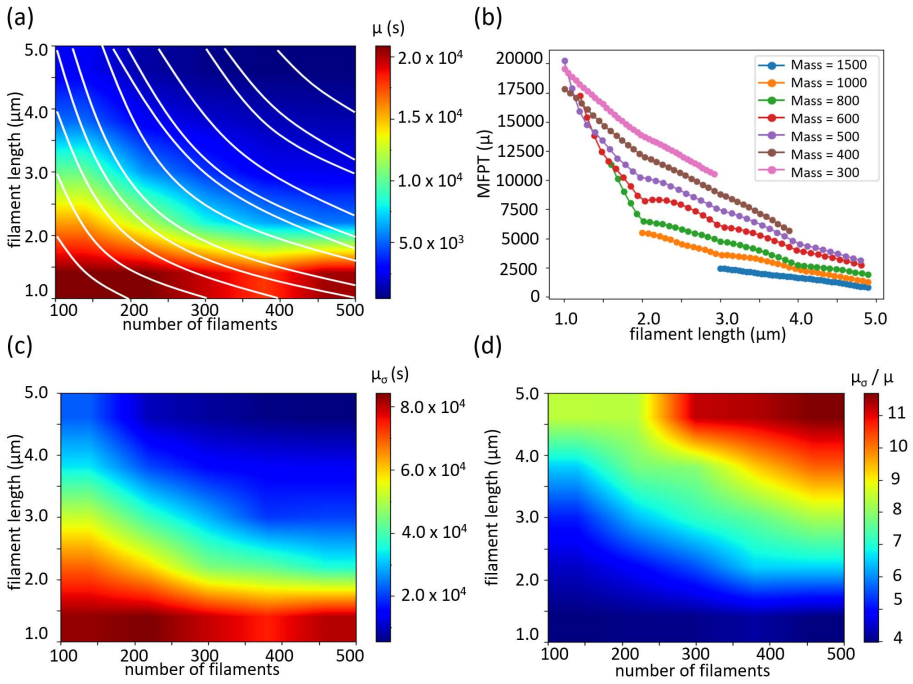

Figure 4.5: (a) MFPT as a function of filament length and number with lines of constant mass in white. (b) MFPT as a function of filament length for different filament masses. Network averaged MFPT standard deviation (c) and normalized average standard deviation (d) as a function of filament length and number. Averaging is over 400 cargo and 100 different networks.

monitor insulin flux out of the cell by making first-passage time distribution (FPTD) measurements for different network parameters.

Fig. 4.4a shows FPTDs as a function of time for different filament lengths with a constant filament number of 300. It should be noted that we calculated the FPTD by binning the first passage times of cargo (starting from a random position with a linearly decreasing probability with distance from the center) when they reached the membrane into 1s time intervals bins. Our simulations have 100 cargos across 100 networks, which makes a total of 10^4 cargos. While the bins go out to 10^6 seconds, in Fig.4.4a the FPTD plots are cut off at 1000 seconds. We notice that at the shortest filament lengths, the FPTD appears to have no peak. The first phase, the initial spike, is only apparent at a filament length of 3 μm and beyond. Thus the filament length clearly tunes this phase that occurs at early times. This is also consistent with our picture, from the previous section, that the early time dynamics are controlled by filament length. Interestingly, it appears that all curves also show a sustained release at late times signified by the long tail. Our results from the previous section suggest

that this second phase at late times is likely a power law decay determined by the value of α. To examine this possibility, we focus on the FPTD behavior at late times. Fig. 4.4b displays a log-log plot of FPTD as a function of time for a network with 300 filaments with a length of 5 μm each, but for different values of α. The larger the value of α, the steeper the decay, until in the case of $\alpha = 1$, the decay is qualitatively different and, becomes exponential. To test whether this second phase can be tuned by the network geometry, we examine, in Fig. 4.4c, the FPTD power law exponent in the second, decaying, transport phase as a function of the filament length and number. We see that the exponent increases with network mass with a more sensitive dependence on filament length. The increase in the exponent is quite significant, from 0.2 to 1.2 in the range of filament parameters studied, indicating that, even though we are looking at relatively late times, the filament network can be used to tune the behavior in that phase too. To analyze this further, we plot, in Fig. 4.4d, the FPTD decay exponent as a function of filament length for several different total filament masses. We see a separation between different mass curves indicating a dependence on the total mass as well as the filament length, with increases in both leading to a larger exponent indicating a steeper decay i.e. a curtailment of the sustained release phase.

Finally, we note that prior work on transport over explicit filament networks in a normally diffusive bulk produced trapping regions that significantly impacted the mean first passage times (MFPT) [31] and also produced a significant variance in MFPT from network to network. To examine whether a similar effect occurs in the presence of cytoplasmic subdiffusion, we measured the MFPT from the FPTDs generated. Fig. 4.5a shows the MFPT (μ) as a function of filament number and length, while fig. 4.5b shows the dependence on filament length for fixed filament mass. Here, μ denotes the average MFPT over all of the network configurations, as each network has its own associated MFPT. As expected, the MFPT decreases with increasing filament mass and filament length, indicating that filaments provide a super-diffusive boost to transport. To examine the effects of filament geometry on transport, we calculate how the MFPT varies across multiple networks. We first calculated the standard deviation for 400 cargo first-passage times on one network and then averaged them across 100 different networks to obtain the network averaged standard deviation, μ_σ. Fig. 4.5c shows μ_σ as a function of filament length and number. We notice that the variance decreases with increasing filament mass indicating that the superdiffusive phase introduced by the filaments works to counteract the variance from the CTRW in the bulk. Also of interest are the rather large values of the normalized average standard deviation (Fig. 4.5d), which is the network averaged standard deviation divided by the MFPT obtained at each particular set of filament parameters. This means that any MFPT variation across networks is dominated by the randomness of the CTRW which overcomes any variations caused by trapping regions due to changes in filament orientation. It is the variance μ_σ that gives rise to the sustained release phase and thus, we see again that a decrease in the filament network mass results in increasing μ_σ and hence an increased sustained release.

4.6 Discussion and Conclusion

In our studies, we have shown that motor-driven transport along filaments is most dominant at early times, as we find in our MSD calculations, where it is apparent that cargos move via superdiffusion. As we change network parameters, namely the filament length and filament number, we can tune this superdiffusive behavior. Increasing the net filament mass, increases the superdiffusive exponent speeding up the transport process and for networks with the same mass, those with longer filaments facilitate even faster transport. The superdiffusion we see in the presence of filaments and the subdiffusion that begins to manifest as filament mass is decreased is consistent with the results found in [39], where α was measured to be about 1.5 in extract, but when the filaments are depolymerized, α decreased to between 0.65 and 0.98. In our simulations, we achieve (Fig. 4.2c) an α value of around 1.5 at a filament length between 2 μm to 3 μm. As we shorten our filaments, the short time exponent drops and transport turns over to the late time regime where CTRW dominates with an α of about 0.8 in the absence of filaments. It is to be noted that this value of α is also consistent with the results from insulin granule subdiffusion in cells treated with vinblastine (a microtubule depolymerizing agent) [4]. There they found that the correlated component of the walk (FBM), was limited to very early times ($\lesssim$10s) and that the process was mostly dominated by CTRW with an $\alpha = 0.8$. It is also interesting to note that their measurements of the TA-MSD exponent overall (in the absence of vinblastine) had a wide spread from subdiffusive to superdiffusive. Our results suggest that, in any such experiment, one could potentially observe a transition from a superdiffusive to a subdiffusive phase as a function of time, or even spatial location, if the network structure is heterogeneous. Thus our simulations of transport over explicit networks coupled to subdiffusion (CTRW) in the bulk highlight regimes where one or the other phase is dominant and quantitatively explains experimentally observed features.

While the role of the cytoskeleton in insulin secretion has not yet been fully understood [53], it is clear that both the cortical actin and microtubule networks are important for the process. It is also clear that there is certainly a reorganization of F-actin upon glucose stimulation that plays a key role. There has been debate about whether the reorganization acts as a removal of a barrier for the granules or a release of trapped granules and how that fits in with results that indicate myosin-powered motility of the granules along F-actin is also important. In our examination of insulin transport, we found that filament length has an important effect on both the early "spike" phase and the second, power-law decay phase in the "biphasic" FPTD. Of particular interest here is that, for networks with shorter filaments, the power-law tail of the distribution is wider, meaning the second phase is maintained for longer. Thus a filament network can contribute to both the early time fast release and upon subsequent shortening also allow the CTRW process to provide a sustained release phase. It is worth noting here that short actin fragments may indeed contribute significantly to the trapping and hence complete depolymerization (i.e. conversion to G-actin) can

have the effect of abolishing CTRW resulting in a comparatively fast release that is not sustained. This is consistent with the fact that glucose stimulation does not alter the F-actin to G-actin ratio and only results in shortening and reorganization [53].

Finally, we showed that, in the presence of a anomalously subdiffusive bulk phase, network to network variation in transport times is less significant than cargo to cargo transport variation over a single network. This suggests that fine-tuned control of the network geometry (to avoid particularly poorly oriented networks) may not be as important in the presence of anomalous subdiffusion in the bulk. While transport as a whole is slower with a higher variance (which can be functional, as in a sustained release), it may be advantageous for the cell in that it may be easier to control quantities such as the filament length and number using regulatory proteins [58,59] than it would be to control filament network arrangements in geometries that limit variation in cargo transport. Taken together, our results suggest that the coupling between superdiffusive and subdiffusive transport modes allow for filament morphology to be used as a control knob to tune transport dynamics *in vivo*.

4.7 Supplementary Information

4.7.1 Anomalous diffusion: Continuous-Time Random Walk (CTRW) in detail

A CTRW is a random walk where steps are taken at random times [40]. However, in our case the step distance is fixed. We choose this particular model of a random walk in order to represent anomalous diffusion. Recall from before, that a characteristic of diffusion is that the mean squared displacement of the random walker as a funtion of time varies with time linearly

$$\langle r^2(t)\rangle \sim t \tag{4.6}$$

However for anomalous diffusion, the mean squared displacement varies with t as:

$$\langle r^2(t)\rangle \sim t^\alpha \tag{4.7}$$

Anomalous diffusion with $0 < \alpha < 1$ is considered subdiffusion. Passive transport within the cytoplasm is thought to be a subdiffusive process.

In our CTRW model, we consider a power-law distribution of wait times. We will use the distribution

$$\psi(t) = \begin{cases} 0 & \text{if } t < 1, \\ \alpha t^{-\alpha-1} & \text{if } t \geq 1. \end{cases} \tag{4.8}$$

where t is the wait time (time step). The probability that the waiting time between steps is greater than t is

$$\Psi(t) = \int_t^\infty \psi(t')dt' \tag{4.9}$$

Additionally, define $\psi_n(t)$ as the probability density that the nth jump occurs at time t. This means that $\psi_1(t) = \psi(t)$ and, because the waiting time between steps is independent

$$\psi_2(t) = \int_0^t \psi_1(t')\psi(t-t')dt' \tag{4.10}$$

Which leads to the general relation

$$\psi_{n+1}(t) = \int_0^t \psi_n(t')\psi(t-t')dt' \tag{4.11}$$

We want to show that our distribution of choice results in subdiffusive behavior. To ensure this recall that we must establish a certian mean squared displacement dependency on t, particularly,

$$\langle r^2(t)\rangle \sim t^\alpha \tag{4.12}$$

In terms of the probability of being at position $\vec{r}$ at time t, this quantity can be determined by

$$\langle r^2(t)\rangle = \int_0^\infty r^2 P(\vec{r},t)d\vec{r} \tag{4.13}$$

We can break up $P(\vec{r},t)$ as follows:

$P(\vec{r},t) = P(\vec{r}) \times$ (probability of remaining at $\vec{r}$ until time t)

Where $P(\vec{r})$ is the probability of being at position $\vec{r}$ regardless of t and

$$P(\vec{r}) = \sum_{n=0}^{\infty} P_n(\vec{r}) \tag{4.14}$$

With $P_n(\vec{r})$ being the probability of being at position $\vec{r}$ at the nth step. Then for $P(\vec{r},t)$,

$$P(\vec{r},t) = \sum_{n=0}^{\infty} P_n(\vec{r}) \int_0^t \psi_n(t')\Psi(t-t')dt' \tag{4.15}$$

We can simplify this expression by first taking the laplace transform:

$$P(\vec{r},t) = \sum_{n=0}^{\infty} P_n(\vec{r}) \int_0^\infty (\int_0^t \psi_n(t')\Psi(t-t')dt')e^{-st}dt \tag{4.16}$$

Using a look-up table, we arrive at the following result

$$P(\vec{r},s) = \sum_{n=0}^{\infty} P_n(\vec{r})\hat{\psi}_n(s)\Psi(s) \tag{4.17}$$

where $\hat{\psi}_n(s)$ and $\Psi(s)$ are the Laplace transforms of $\psi_n(t)$ and $\Psi(t)$ respectively. We can determine these values in terms of the Laplace transform of our original wait time distribution, $\psi(t)$. Investigating separately, making use of our previous definitions of $\psi_n(t)$ and $\Psi(t)$,

$$\hat{\psi}_n(s) = \int_0^\infty (\int_0^t \psi_{n-1}(t)\psi(t-t')dt')e^{-st}dt \tag{4.18}$$

$$= \hat{\psi}_{n-1}(s)\hat{\psi}(s) \tag{4.19}$$

Recursively solving the problem of $\hat{\psi}_{n-i}(s)\hat{\psi}(s)$ from $i = 1$ to $i = n-1$,

$$\hat{\psi}_n(s) = \hat{\psi}(s)^n \tag{4.20}$$

For $\Psi(t)$, the Laplace transform is

$$\hat{\Psi}(s) = \int_0^\infty \Psi(t)e^{-st}dt \tag{4.21}$$
$$= \int_0^\infty (\int_t^\infty \psi(t')dt')e^{-st}dt \tag{4.22}$$
$$= \int_0^\infty (1 - \int_0^t \psi(t')dt')e^{-st}dt \tag{4.23}$$
$$= \frac{1-\hat{\psi}(s)}{s} \tag{4.24}$$

We then have

$$\hat{P}(\vec{r}, s) = \sum_{n=0}^{\infty} P_n(\vec{r})\hat{\psi}(s)^n \frac{1-\hat{\psi}(s)}{s} \tag{4.25}$$

We can begin to eliminate the infinite sum by taking the Fourier transform

$$\hat{P}(\vec{k}, s) = \sum_{n=0}^{\infty} (\int_{-\infty}^{\infty} e^{-i\vec{k}\cdot\vec{r}} P_n(\vec{r})d\vec{r})\hat{\psi}(s)^n \frac{1-\hat{\psi}(s)}{s} \tag{4.26}$$

Where we have, similar to the case for $\psi_{n+1}(t)$,

$$P_n(\vec{r}) = \int P_{n-1}(\vec{r'})P(\vec{r} - \vec{r'})d\vec{r'} \tag{4.27}$$

We then have the Fourier transform:

$$\hat{P}_n(\vec{k}) = \int_{-\infty}^{\infty} e^{-i\vec{k}\cdot\vec{r}} P_n(\vec{r})d\vec{r} \tag{4.28}$$
$$= \int P_{n-1}(\vec{r'})d\vec{r'} \int_{-\infty}^{\infty} P(\vec{r} - \vec{r'})e^{-i\vec{k}\cdot\vec{r}}d\vec{r} \tag{4.29}$$
$$= \int P_{n-1}(\vec{r'})\hat{P}(\vec{k})e^{-i\vec{k}\cdot\vec{r'}}d\vec{r'} \tag{4.30}$$
$$= \hat{P}_{n-1}(\vec{k})\hat{P}(\vec{k}) \tag{4.31}$$

where $\hat{P}(\vec{k})$ is the Fourier transform of $P(\vec{r})$. Similar to before, by solving the above problem recursively,

$$\hat{P}_n(\vec{k}) = \hat{P}(\vec{k})^n \tag{4.32}$$

Then,

$$\hat{P}(\vec{k}, s) = \frac{1-\hat{\psi}(s)}{s} \sum_{n=0}^{\infty} \hat{P}(\vec{k})^n \hat{\psi}(s)^n \tag{4.33}$$

Assuming $\hat{P}(\vec{k})$ and $\hat{\psi}(s)$ are normalized, then for any $\vec{k}$ and s, $|\hat{P}(\vec{k})\hat{\psi}(s)| \leq 1$. Meaning that the infinite series converges. We the have

$$\hat{P}(\vec{k}, s) = \frac{1 - \hat{\psi}(s)}{s} \frac{1}{1 - \hat{\psi}(s)\hat{P}(\vec{k})} \tag{4.34}$$

In moving towards our goal of determining subdiffusive behavior, we need to determine

$$\langle r^2 \rangle = \int_{-\infty}^{\infty} r^2 P(\vec{r}) d\vec{r} \tag{4.35}$$

Then as a function of s, we have

$$\langle r^2(s) \rangle = \hat{\psi}(s)^n \frac{1 - \hat{\psi}(s)}{s} \int_{-\infty}^{\infty} r^2 P_n(\vec{r}) d\vec{r} \tag{4.36}$$

Which, based on our previous calculations, means that

$$\langle r^2(s) \rangle = -\frac{\partial^2}{\partial \vec{k}^2} \hat{P}(\vec{k}, s)|_{\vec{k}=0} \tag{4.37}$$

Performing the differentiation:

$$\frac{\partial^2}{\partial \vec{k}^2} \hat{P}(\vec{k}, s)|_{\vec{k}=0} = \frac{(1 - \hat{\psi}(s))\hat{\psi}(s)}{s(1 - \hat{\psi}(s)\hat{P}(\vec{k}))^2} \frac{d^2 \hat{P}(\vec{k})}{d\vec{k}^2}|_{\vec{k}=0} \tag{4.38}$$

$$+ \frac{(-2)(-\hat{\psi}(s)^2(1 - \hat{\psi}(s)))}{s(1 - \hat{\psi}(s)\hat{P}(\vec{k}))^3} (\frac{d\hat{P}(\vec{k})}{d\vec{k}})^2|_{\vec{k}=0} \tag{4.39}$$

Recall that $\hat{P}(\vec{k})$ is the Fourier transform of the probability distribution of step sizes. So,

$$\hat{P}(\vec{k})|_{\vec{k}=0} = \int_{-\infty}^{\infty} P(\vec{r'}) d\vec{r'} = 1 \tag{4.40}$$

$$\frac{d\hat{P}(\vec{k})}{d\vec{k}}|_{\vec{k}=0} = -i \int_{-\infty}^{\infty} \vec{r'} P(\vec{r'}) d\vec{r'} = -i\langle \vec{r'} \rangle \tag{4.41}$$

$$\frac{d^2 \hat{P}(\vec{k})}{d\vec{k}^2}|_{\vec{k}=0} = -\int_{-\infty}^{\infty} (\vec{r'})^2 P(\vec{r'}) d\vec{r'} = -\langle (\vec{r'})^2 \rangle \tag{4.42}$$

For a fixed step size, let $\vec{r'} = \delta\vec{r}$. Then, because the step can be in any direction,

$$\langle \delta\vec{r} \rangle = 0 \tag{4.43}$$

And we have that

$$\langle r^2(s)\rangle = \frac{\hat{\psi}(s)}{s(1-\hat{\psi}(s))}(\delta\vec{r})^2 \tag{4.44}$$

We can then determine the dependence of $\langle r^2(s)\rangle$ if we can get the Laplace transform of $\psi(t)$, our original wait time distribution.

Recall that we had

$$\psi(t) = \begin{cases} 0 & \text{if } t < 1, \\ \alpha t^{-\alpha-1} & \text{if } t \geq 1. \end{cases} \tag{4.45}$$

The Laplace transform of this is then

$$\hat{\psi}(s) = \int_1^\infty e^{st}\alpha t^{-\alpha-1}dt \tag{4.46}$$

Performing an integration by parts:

$$\int_1^\infty e^{st}\alpha t^{-\alpha-1}dt = -e^{-st}t^{-\alpha}|_1^\infty - s\int_1^\infty e^{-st}t^{-\alpha}dt \tag{4.47}$$

Breaking up the integral on the far right into a difference of two integrals:

$$\int_1^\infty e^{-st}t^{-\alpha}dt = \int_0^\infty e^{-st}t^{-\alpha}dt - \int_0^1 e^{-st}t^{-\alpha}dt \tag{4.48}$$

The first term on the right-hand side is a Laplace transform with the following solution:

$$\int_0^\infty e^{-st}t^{-\alpha}dt = \frac{\Gamma(-\alpha+1)}{s^{-\alpha+1}} \tag{4.49}$$

We now have

$$\hat{\psi}(s) = -e^{-st}t^{-\alpha}|_1^\infty + s\int_0^1 e^{-st}t^{-\alpha}dt - \Gamma(-\alpha+1)s^\alpha \tag{4.50}$$

For large t, $s \to 0$. So now, keeping leading order terms:

$$\hat{\psi}(s) \approx 1 - \Gamma(-\alpha+1)s^\alpha \tag{4.51}$$

Which means that

$$\langle r^2(s)\rangle = \frac{\hat{\psi}(s)}{s(1-\hat{\psi}(s))}(\delta\vec{r})^2 \tag{4.52}$$

$$\approx \frac{1-\Gamma(-\alpha+1)s^\alpha}{s(1-(1-\Gamma(-\alpha+1)s^\alpha))} \tag{4.53}$$

$$= \frac{1-\Gamma(-\alpha+1)s^\alpha}{\Gamma(-\alpha+1)s^{\alpha+1}} \tag{4.54}$$

$$\tag{4.55}$$

The behavior if this expression roughly depends on its leading-order term:

$$\langle r^2(s)\rangle \approx \frac{1}{\Gamma(-\alpha+1)s^{\alpha+1}} \tag{4.56}$$

To get the dependence on t, we take the inverse Laplace transform of this which results in:

$$\langle r^2(t)\rangle \approx \frac{1}{\Gamma(-\alpha+1)\Gamma(\alpha+1)}t^\alpha \tag{4.57}$$

Which means we have the dependence:

$$\langle r^2(t)\rangle \sim t^\alpha \tag{4.58}$$

This result means that our CTRW model results in anomalous subdiffusion.

4.7.2 MSD calculations at a smaller cell radius (10 μm)

In our preliminary MSD calculations, we maintained a cell radius of 10 μm. A consequence of this is that we see greater effects of the confinement on the subdiffusive behavior at later times rather just the effects of the CTRW. This is a primary reason why we see $\alpha < 0.8$ at relatively later times and why we move the cell radius out to 20 μm. This is demonstrated in Fig. 4.6

4.7.3 System parameters

Shown in Table 4.1 are the common parameter values for our system. We have used most of them in previous work [31]. The subdiffusive exponent that we use ($\alpha = 0.8$) is close to physical [39, 41].

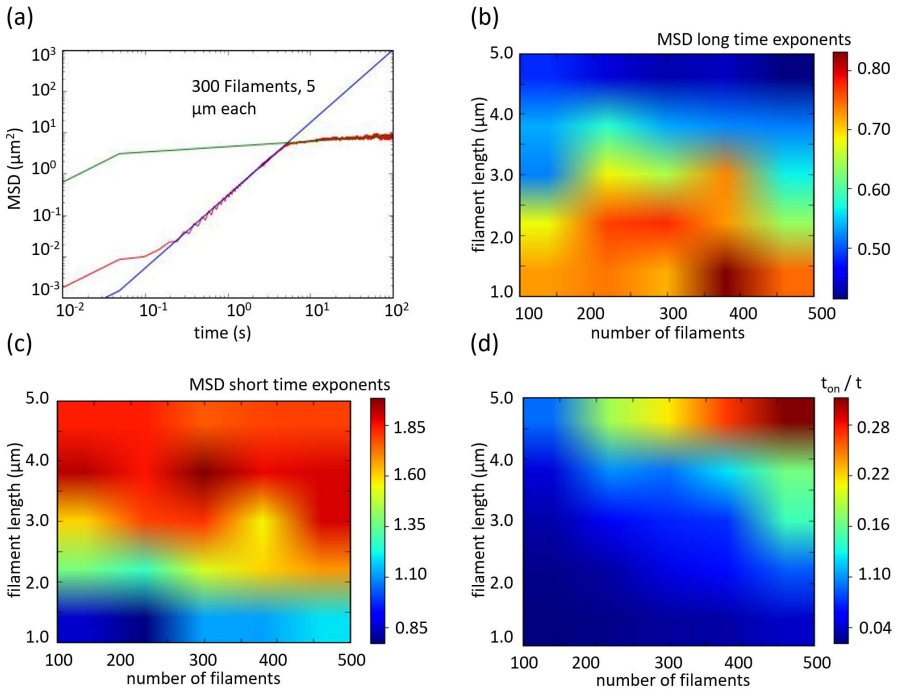

Figure 4.6: (a) MSD as a function of time for transport of cargos over 300 filaments, each with a length of 5 μm. Each cargo begins near the nucleus of a cell with a radius of 10 μm. (b) Due to greater confinement than in the case of a 20 μm cell, the "long-time" exponents reach smaller values ($0.5 < 0.8$). (c) In the short time, with $\alpha > 1.0$, we still see an indication of superdiffusion (at least for filaments greater than 2 μm). (d) As one would expect, given that the cargo attachment and detachment rates stay constant during all simulations, the fraction of the time spent on the network increases with aggregate filament length ($Number of filaments \times Filament Length$).

Table 4.1: Table of System Parameters and Values

Parameter	Value (and source of that value)
Cell Radius	10 μm [31]
Nuclear Radius	5 μm [31]
Cargo Radius	100 nm [31]
Cargo Attachment Rate (k_{on})	5 s^{-1} [31]
Cargo Detachment Rate (k_{off})	1 s^{-1} [31]
Diffusion Constant (D)	0.051 $\mu m^2/s$ [31]
Cargo Movement Speed on Filaments	1 $\mu m/s$ [31]
CTRW Exponent (α)	0.8 [39, 41]

Chapter 5

Real Network Simulations

5.1 Introduction

When cargos have multiple motors attached to them, cargos are able to switch on to different filaments while in the active phase of motion when they approach an intersection of multiple filaments [6, 15, 60].

To study the effect of motor number on how often cargos switch to different filaments at filament intersections, we simulate the movement of cargos on networks of filaments using a methodology similar to what we've done previously [31]. To help in understanding how the cargo switching probability relates to the number of motors attached to each cargo, we compare our simulation results with cargos whose movement was tracked experimentally as each cargo was carried by kinesin motors along bundles of microtubules. There are two sets of experimental data that we compare our results to. The positions of cargos with one kinesin motor attached to them were tracked as well as those with ten kinesin motors attached to them. To run our simulations and make a comparison with this data set, we use filaments obtained from an image of one of these networks of microtubule bundles that were used to provide for the cargos' movement. From here, we attempt to better understand the relationship between the number of motors attached to a cargo and that cargo's probability of switching on to another filament when it is at a filament intersection in order to provide a foundation to develop further theoretical models for how cargos move along real filament networks. All experimental data is provided by the J. L. Ross lab at the University of Massachusetts, Amherst.

5.2 Implementing the FIRE algorithm and determining run lengths of different cargo detachment rates

Before running cargo simulations, we need to extract a filament network from an image. The FIbeR Extraction (FIRE) algorithm [61] takes care of this. Its input is

an image of an actual network of filaments. The image we use is an image consisting of a network of microtubule bundles. We will consider each bundle that the FIRE algorithm picks up to be one filament. The FIRE algorithm assigns a number to each filament, as well as each so-called vertex that makes up a filament. That is, Each filament is made up of numerous line segments whose endpoints are called vertices. So the FIRE algorithm outputs the filament numbers, the vertex numbers that belong to each filamant, and also the x and y positions of each vertex in terms of pixel number.

With this information, we make calculations to gain additional information for our own purposes. Firstly, we convert each x and y value corresponding to a vertex position from a pixel number to a μm value. This way, we can use our standard values of a diffusion constant of $D = 0.051\mu m^2/s$ and cargo movement speed of $v = 1\mu m/s$ to govern cargo movement. The resolution of the image we use is $0.0675\mu m/pixel$. We also keep track of which vertices are a part of multiple filaments. This gives us the positions of filament intersections which is important in getting information about how often cargos switch to different filaments.

Once we all of the network information we need, we can begin running simulations of cargos. Cargos begin diffusing off filaments until they approach within 100 nm of one. When they are within range of a filament, cargos bind with a rate of k_{on}. For our purposes, we have cargos immediately bind to filaments once they are near one. Cargos move along filaments until they fall off. The rate at which cargos detach from filaments, k_{off} corresponds to the number of motors that would be attached to the cargo. The more motors there are attached to the cargo, the less likely it would detach completely from the filament it is walking on.

With the network information we have about the image in Fig. 5.1a, whose filaments obtained from the FIRE algorithm are colored in Fig. 5.1b, we are able to simulate the movement cargos. As each cargo attaches to a filament, moves along it, then falls off, we keep track of each cargo's run length. The run length is inversely proportional to the cargo off rate. Our goal here is to compare run length distributions obtained through simulations with those that were obtained experimentally. In Fig. 5.1c, we show CDF curves for cargo run lengths obtained from cargos whose movements were tracked experimentally. Two sets of cargos were tracked experimentally: those with one kinesin motor attached to them, and those with ten kinesin motors attached them. We fit each of the one- and ten-motor cargos' run length CDFs to exponential functions. In Fig. 5.1d, we show the CDFs and their fits for two sets of 1000 cargos' simulated trajectories and give the exponential functions that were used to fit these two sets of data. We attempt to match the fits of the CDFs for the experimental data. We were able to match the one-motor experimental fit using an off rate of $k_{off} = 0.1s^{-1}$ and we were able to match the ten-motor experimental fit by using an off rate of $k_{off} = 0.01s^{-1}$. This gives us an estimate for the off rates of one- and ten-motor cargos, respectively. When computing the cargo trajectories and run lengths, we imposed a cargo switching probabilty of 0.0. That is, if a cargo, while moving along a filament, reaches a known filament intersection, there is a 0% chance that it will switch to another filament.

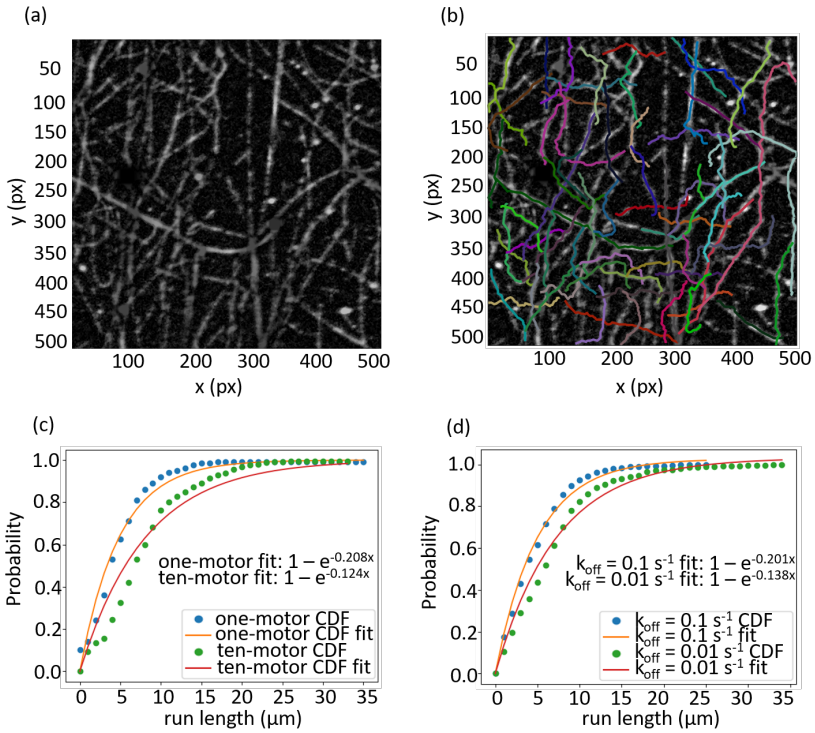

Figure 5.1: (a) Original image of the microtubule network. (b) The extracted network after implementing the FIRE algorithm. Each filament is colored differently. (c) Run length CDFs for cargos tracked experimentally. The two different CDFs correspond to the run lengths obtained from trajectories from cargos attached to one kinesin motor and cargos attached to ten kinesin motors. The rates that the CDF curves increase relate to the rate at which motors detach the microtubules. (d) Run length CDFs obtained from simulations of cargos moving on the networks extraced from the FIRE algorithm. Off rates are tuned until the CDF fits approach the obtained from fitting the experimental data in (b). The off rate associated with one motor cargos is approximately $k_{off} = 0.1s^{-1}$ and the off rate associated with ten motor cargos is approximately $k_{off} = 0.01s^{-1}$. This is the off rate we use when making the rest of our calculations.

5.3 Calculating cargo trajectories and ensemble average MSDs

Cargos begin moving, via diffusion, off of filaments. When a cargo gets within 100 nm of a filament, it attaches to it. Once attached, the cargo will move either backward or forward along the filament with equal probability. The forward direction of movement is defined by the list of vertices that make up each filament as determined by implementing the FIRE algorithm. For example, if the cargo attaches to filament number 7, which is made up of the given vertices 2, 3, 4, 10, 14, and 15, in that order,

the forward direction will then be the path of motion moving towards the vertices 2, then 3, then 4, and so on up to vertex 15. If the cargo reaches the end of the filament (the last vertex in the list of vertices that make up the filament) without detaching or switching to another filament, the cargo will "walk" off the end of the filament. If the cargo reaches within 100 nm of a filament intersection, which would be a vertex that is shared by more than one filament, the cargo will switch to the other filament(s) with a given switching probability and then begin to move forward or backward along that filament. When a cargo detaches or walks off the end of a filament, the simulation of its movement ends. As each cargo moves along the filament network, its position is tracked so its trajectory is traced out.

Fig. 5.2 shows four sets of 1000 cargo trajectories for different switch probability values. The path that each cargo moves is initially traced out in blue. The tortuosity (τ) of a path is defined as the ratio of the length of a path (L) to the distance between bath endpoints (D),

$$\tau = \frac{L}{D}. \tag{5.1}$$

For cargo paths in Fig. 5.2 that have $\tau > 2$, their path is colored yellow. Notice that cargos are more likely to travel paths with higher tortuosity values if there is greater probability that they will switch to other filaments at filament intersections. However, even if a cargo never switches to other filaments, the tortuosity of its path will likely be greater than one because the filaments that are extracted using the FIRE algorithm are usually not perfectly straight.

To test how well our simulations match up with the movement of actual cargos with either one motor attached to them or ten motors attached to them, it is useful to calculate the ensemble average MSD for cargos at different switching probability values and compare them the the MSDs for the one- and ten- motor cargos that were tracked experimentally. To extract necessary information from the MSD plot, we must use a proper fit, and extract useful parameters. The fit we use was developed in [62], where bacteria are treated as random walkers that move at contant velocity v before turning at one of two preferred angles, $\Delta\phi_1$ or $\Delta\phi_2$ at a "tumbling rate" λ. The MSD is determined to be

$$\begin{aligned}\langle[\vec{r}(t) - \vec{r}(0)]^2\rangle =& \frac{v^2}{\lambda^2(1-\alpha\beta)^2}[(1-\alpha\beta)(2+\alpha+\beta)\lambda t - 2(1+\alpha)(1+\beta) \\ &+ e^{-\lambda t}\{\frac{\alpha+\beta+\alpha\beta(4+\alpha+\beta)}{\sqrt{\alpha\beta}} sinh(\sqrt{\alpha\beta}\lambda t) \\ &+ 2(1+\alpha)(1+\beta)cosh(\sqrt{\alpha\beta}\lambda t)\}]\end{aligned} \tag{5.2}$$

where,

$$\alpha = \langle cos\Delta\phi_1\rangle, \tag{5.3}$$

$$\beta = \langle cos\Delta\phi_2\rangle. \tag{5.4}$$

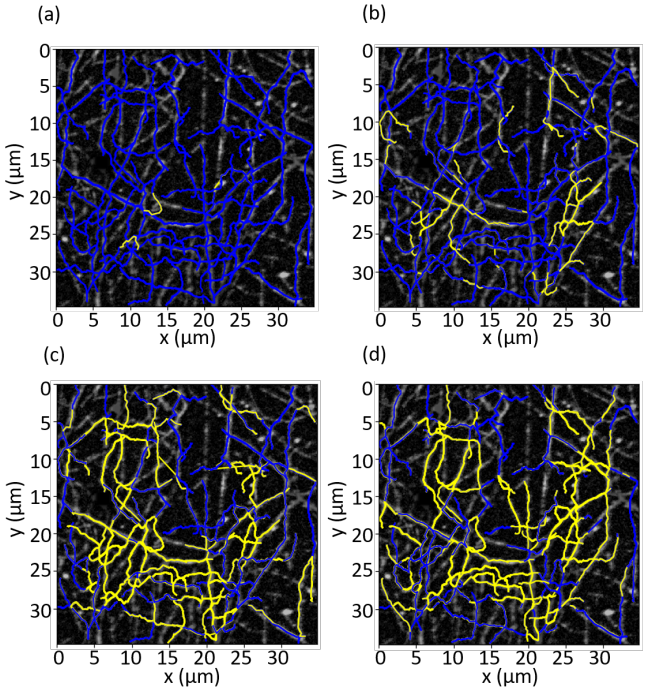

Figure 5.2: (a) trajectories of cargos traveling on the FIRE network. The probability of switching to another filament at filament intersections is 0.0. The cargo paths are colored blue. The paths with a tortuosity greater than two are colored yellow. (b) cargo trajectory paths when the switching probability is 0.3. (c) trajectories of cargos that have a switching probability of 0.7. (d) trajectories of cargos that have a switching probability of 1.0.

For our purposes, in order to properly fit this to our data and extract the parameters we want, we assume cargos turn at an average angle of $p\pi/2$, so that,

$$\alpha = \beta = cos(p\pi/2), \tag{5.5}$$

and we define the quantity d, where

$$\lambda = \frac{v}{d}, \tag{5.6}$$

and in our simulations, $v = 1\mu m/s$.

So when we extract parameters from this fit, for our simulations, we extract the "effective" distance between filament intersections, d, and "effective" cargo switching probability, p.

We apply the fit for the MSD to the calculated ensemble average MSD for 1000 cargos at different switching probabilities in Fig. 5.3a. For each of these fits, we extract the parameters d and p and plot them as a function of the simulation-imposed switching probability in Fig. 5.3b. Because the fit can be applied at all timescales, we focus on the short timescales (up to ten seconds) because cargos have not yet begun

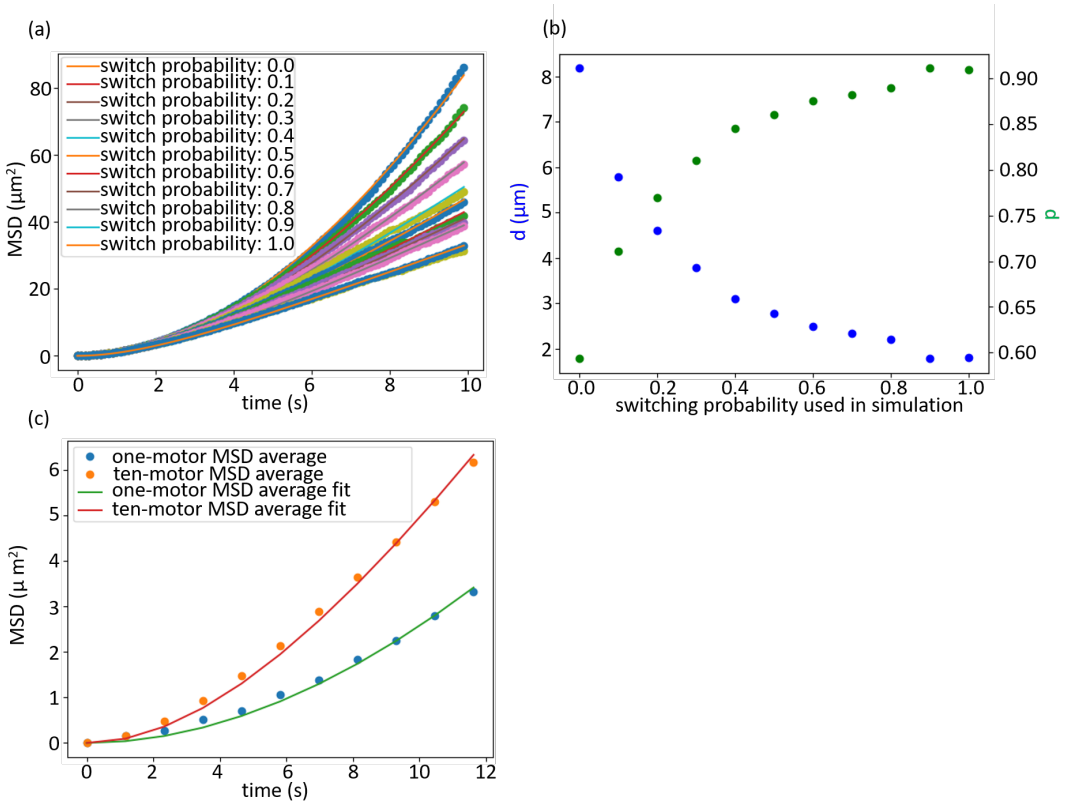

Figure 5.3: (a) ensemble average MSDs for 1000 cargos for different switching probabilities. The average MSD increases faster for smaller switching probability values. (b) parameters, d ("effective" distance between filament intersections) and p ("effective" switching probability) extracted from the MSD fits, plotted as a function of the switching probability used in the simulations. There is what we call an "effective" switching probability and distance between filament intersections because the filaments used in the simulations are not perfectly straight. For higher imposed switching probabilities, it is as if the cargos are switching more often and in shorter distance increments. (c) ensemble average MSDs plotted for the one- and ten-motor cargo trajectories obtained experimentally. The curve fits are plotted as straight lines. With these fits, we extract the parameters $v = 0.17\mu m/s$, $d = 4.72\mu m$, and $p = 0.99$ for the one-motor cargos, and $v = 0.27\mu m/s$, $d = 2.09\mu m$, and $p = 0.99$ for the ten-motor cargos.

to more "regularly" detach from and walk off of filaments, meaning we can get better averaging at short times.

Fig. 5.3c shows the MSD fit applied to the calculated ensemble average MSD for the experimentally-tracked one-motor cargos and ten-motor cargos. These cargos actually travelled, on average, slower than the cargos in our simulations, but in extracting the parameter d from these fits, we actually see that the "effective" distance between filament intersections is d = 4.72 μm for the one-motor cargos and d = 2.09

μm for the ten-motor cargos. From this perspective, one-motor cargos travel twice as far as the ten-motor cargos before "switching" to another filament. One thing that is interesting to note is that for the extracted network, upon learning the locations of filament intersections, we were able to calculate an average distance between filament intersections of 2.413 μm, which is near the approximate value of d for the one-motor cargos and the imposed probability of 0.0 used in the simulations (near 2 μm).

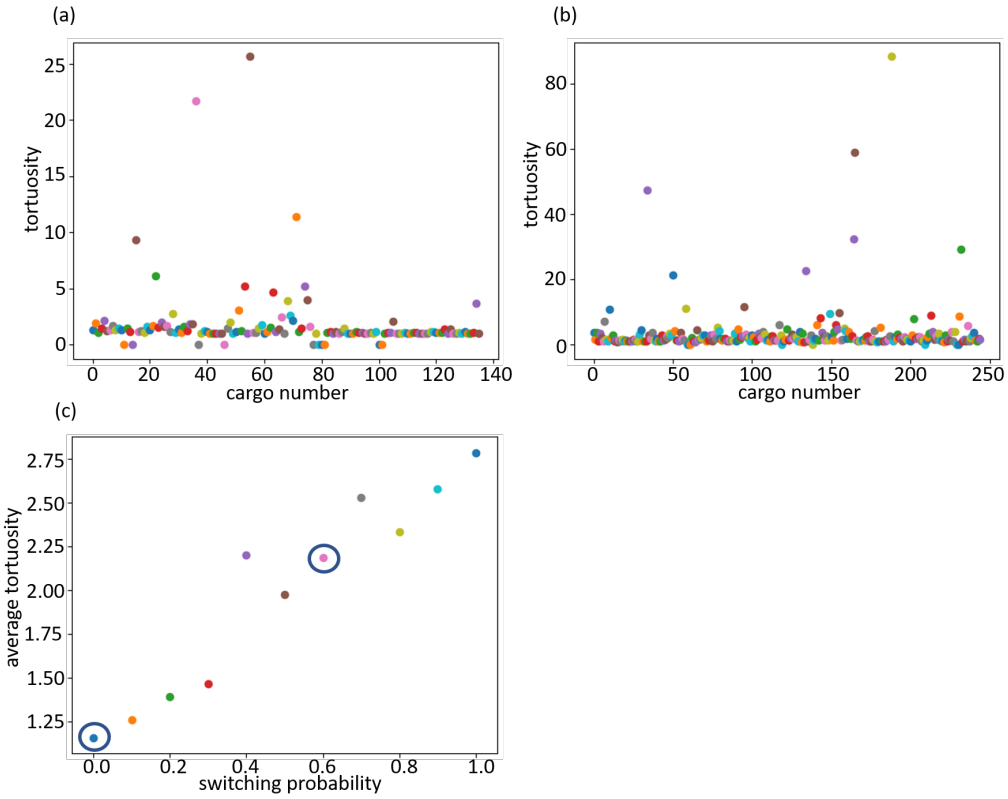

Figure 5.4: (a) Tortuosity of each of the one-motor cargo's experimental trajectory. The standard deviation of these tortuosities is 3.036. To cut the outliers we eliminating the tortuosities above one standard deviation. After doing this, the average tortuosity is 1.186. (b) Tortuosity of each of the ten-motor cargo's experimental trajectory. The tortuosity standard deviation is 7.964. After cutting values above one standard deviation, the average tortuosity is 2.161. (c) The average tortuosity of the cargo trajectories obtained through simulations, plotted as a function of imposed cargo switching probability. The circled values are the ones which are closest to the average one- and ten-motor cargo trajectory tortuosities. A switching probability of 0.0 corresponds to the average tortuosity of one-motor cargos while a switching probability of 0.6 corresponds to the average tortuosity of ten-motor cargos. These values help quantify how often cargos switch to different filaments at filament intersections depending on how many motors are attached to the cargo.

5.4 Comparing average tortuosities for cargo trajectories obtained from experiments and from simulations

Another way we try to correlate the switching probabilities used in our simulations with the number of motors on cargos that are tracked experimentally, is to compare the average tortuosity of the paths of the tracked one-motor cargos and the paths of the tracked ten-motor cargos with the average tortuosity of the paths of the cargos whose movement is simulated, for different switching probabilities.

Fig. 5.4a shows the tortuosity of each of the experimental one-motor cargo's path. There are some tortuosity values that are likely too large, with values up to just above 25. These are likely outliers due to errors in hand-tracking. What we do then, is cut out the values above one standard deviation from the average tortuosity of all cargos and then calculate a new average. This new average is 1.186 for the experimentally-tracked one-motor cargos. Fig. 5.4b shows the tortuosities for all of the ten-motor cargos. Following the same procedure that we did for the one-motor cargos, the new average tortuosity ends up being 2.161. In Fig. 5.4c, we plot the average tortuosity for 1000 cargos whose movement was simulated, for different switching probabilities. In the plot, we circle two average tortuosity values. One at a switching probability of 0.0 and another at a switching probability of 0.6. These are the average tortuosities that are closest in value to the average tortuosities that were calculated from experimental data. A switching probability of 0.0 will then correspong to cargos with one motor attached and a switching probability of 0.6 will correspond to cargos with ten motors attached. A switching probability of 0.0 for one-motor cargos makes sense because these cargos should not, generally, switch to different filamants, and a switching probability of 0.6 for ten-motor cargos makes some sense, especially when comparing to [60].

5.5 Conclusion and future directions

In comparing the data achieved from our simulations for different imposed cargo switching probabilities at filament intersections with the experimental data set where the movement of one- and ten-motor was tracked, we have established reasonable correlations between cargo switching probabilities and the number of motors attached to the cargo. However, there are opportunities to expand on these results.

One approach that may be taken is calculate the force from each motor on the cargo due to the presence of nearby filaments [15]. The direction the cargo moves and, therefore, the filament it eventually attaches to and moves along, can then be determined by the number of motors attached to it as well as the geometry of the filament intersection. From this perspective, the cargo switching probability will not necessarily be fixed for every filament intersection.

Our particular model can be refined by considering different filament and motor types. Different switching probabilities have been calculated for microtubule-microtubule, microtubule-actin, and actin-actin intersections [6]. We can take these into account by running simulations on networks consisting of both microtubules and actin filmanets where cargos that may contain kinesin, dynein, or myosin motors are able to move.

Chapter 6

Final Discussion

6.1 Conclusions

We have extended our group's work in [31] in three primary directions. By establishing a methodology for integrating cargo distributions, We are able to eliminate the random noise inherent in conducting simulation in our calculations. The survival probability of the distribution is sensitive to the lengths of the filaments that make up the cytoskeletal network as well as each filament's polarization bias. Interestingly, we see that at a filament length of 3 μm and a polarization bias of 0.3, parts of the distribution can become "trapped" in that the orientation of filaments at random locations throughout the cell hinders escape from the outer membrane.

When incorporating anomalous diffusion in the bulk, we see an interplay between superdiffusive and subdiffusive transport. Superdiffusive transport correlates to the active transport phase, when cargos are moving along filaments. Because superdiffusive transport is much faster than subdiffusive transport, Most of the transport at relatively early times is governed by superdiffusion. The resulting coupling between superdiffusive and subdiffusive transport allows the primay filament network parameters such as filament length, filament number, and filament polarization direction to highly tune transport dynamics without too much network-to-network variation.

We were also able to achieve some comparison of our simulation data with experimental data. We took images of actual networks of microtubule bundles, which were used to track movement of cargos, and extracted network information to use in our simulation. With this capability, we were able to obtain a theoretical model for an MSD fit as a function of time for cargos moving along the extracted network, and compare our simulation data with experimental data. In this model, we incorporated cargos with multiple motors attach by imposing a probability of a cargo to switch to another filament at filament intersections in our simulations. Through this mechanism, we found a correlation between switching probabilities used in the simulations with the number of motors attached to cargos in the experiments.

6.2 Future Possibilities

With the work we've done, there is a lot of room for expansion. For example, we don't necessarily have to place filaments randomly. Instead, we can consider a more biologically realistic network where microtubules are oriented radially outward and a relatively thin layer of actin filaments lies near the cell membrane. We can then compare our results, especially our numerical results, with some simulations that have also considered this geometry [42].

Something else to consider is that usually at filament endpoints, cargos have different binding and unbinding rates. Also, cargos may or may not just walk off the end of filaments. Whether or not this happens can depend on the motor type and the filament polarization rate [47]. Another thing to consider is that there may be cargo-crowding at filament endpoints, meaning that in this situation, cargos unbind at higher rates [48].

We can also expand on our attempt at understanding the link between the switching probability of cargos at filament intersections and the number of motors attached to each cargo. One thing we might attempt is to calculate the force from each motor on the cargo when motors are attached to nearby filaments [15]. The direction of cargos movement and the resulting filament it moves along, can then be determined by the number of motors attached to it as well as the geometry of the filament intersection. From this perspective, the cargo switching probability will not necessarily be fixed for every filament intersection. We can also consider different switching probabilities for different motor types and filament intersections. There are different switching probabilities for microtubule-microtubule, microtubule-actin, and actin-actin intersections [6]. We can consider these different probabilities by running simulations on networks consisting of both microtubules and actin filaments where cargos may be attached to kinesin, dynein, or myosin motors.

Another thing to take into consideration is the dimension and shape of the cell. Actual eukaryotic cells are three-dimensional, whereas we primarily consider two-dimensional cells where approximate spherical symmetry is assumed. This means we can further extend our model to a three-dimensional system. Another direction we can go is the consideration of a different cell structure (like an elongated, elliptically shaped cell) and different target destinations for cargos (these might be locations in the cell where varias organelles may reside). Furthermore, filament defects have been observed to affect motor movement, giving us another possibility to consider [21].

Finally, one thing to note is that actual cytoskelatal networks are highly dynamic. The processes of transport itself and the the changing cytoskeletal network occur over similar timescales, taking seconds to minutes [58, 59, 63, 64]. Within the cytoskeleton, both the actin filaments and the microtubules change. To implement this, we can allow each filament in the network to change according to how they have been observed to: each filament will be allowed to polymerize, depolymerize, branch, become capped, and get severed according to the concentrations of actin and regulatory proteins responsible for the changing filaments [58]. If we implement microtubules

into our networks, their dynamics will be similar [63]. Some things that we anticipate happening is the possibility that highly dynamic cells make the active transport phase approach a diffusive process, and the possibility that a changing network could help "untrap" cargos stuck in the trapping regions we had oberved previously in both our simulations and numerical integrations [31]. With the directions we can take with this research, we have not only developed a new theoretical model for intracellular tranport, but have also provided multiple ways to expand on it.

Chapter 7

Appendix: Computer Programs Used

7.1 Introduction

In this work, all simulations and numerical calculations were executed using C or C++ and all resulting data was analyzed using Python. Here, we show the programs used to conduct the different research projects.

7.2 Numerical Calculation Programs

7.2.1 transNetNum.c

This program computes the time evolution of a distribution of cargos until the distribution leaves an area designated as the system's cell.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define N 100000 // number of time steps
#define NX 210 // number of x coordinates
#define NY 210 // number of y coordinates

// global variables

// sets resolution and distance step size and max x and y
   values
```

```
double dx = 0.1, dy = 0.1, xmax, ymax, xc, yc, dx2, dy2;

// arrays to be used in integration of advection diffusion
   equation

// total probability distribution, "off" distribution, "on"
   distribution
double p[NX][NY], pOff[NX][NY], pOn[NX][NY];
double pOnNew[NX][NY], pOffNew[NX][NY];

// "off" and "on" switching rates
double kOff[NX][NY], kOn[NX][NY];

// diffusion constant
double D[NX][NY];

// velocity field arrays
double vx[NX][NY], vx2[NX][NY], vy[NX][NY], vy2[NX][NY];

// indicator for outside inner radius and inside outer radius
double C[NX][NY];

// array that indicates where the filaments are located
double filNet[NX][NY];

// array that indicates where filament endpoints are located
double filEnds[NX][NY];

// array of filament polarities
double pols[NX][NY];

// survival probability
double S[N];

// first passage time distribution
double F[N];

// mean first passage time
// initialize to 0.0
double mfpt = 0.0;

// number of filaments and filament length
int numFil = 150;
double l = 3.0;

// size of time step
```

```
double dt = 0.01;

// how much time has passed
double t;

// radius of inner and outer boundaries
double outer = 10.0, inner = 5.0, outer2, inner2;

// initial (radial) width of the probability distribution
double width = 0.4;

// filament on and off rates and velocity along filaments
double Koff = 0.8, Kon = 4.0, vTot = 1.0;

// bulk diffusion constant
double Db = 0.051;

int main()
{

 // position (i, j)  and time indices (m) for arrays
 int i, j, m;

 // total time elapsed, time spend off filaments, time spent
    on filaments
 double t, tOff, tOn;

 // "hypotenuse" step size
 double dr, dr2;


 // setting initial global variable values

 // max x and y coordinates
 xmax = NX * dx;
 ymax = NY * dy;

 // (x, y) coordinates of the cell's center
 xc = xmax / 2;
 yc = ymax / 2;

 dx2 = dx * dx;
 dy2 = dy * dy;
 dr2 = dx2 + dy2;
 dr = sqrt(dr2);
```

```
// squares of the inner and outer radii
inner2 = inner * inner;
outer2 = outer * outer;

// probability distribution normalization constant
double norm = 0.0;

// set up diffusion and initial probability distribution
   arrays
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  // diffusion constant is the same everywhere
  D[i][j] = Db;
  // probability distribution begins in a "ring", off all
     filaments
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2 &&
      pow((i*dx-xc), 2) + pow((j*dy-yc), 2) < pow((inner+
         width), 2)) {
   pOff[i][j] = 1.0;
  }
  // keep track of normalization constant
  norm = norm + pOff[i][j]*dx*dy;
  // inner cell boundary is reflecting
  // only allow transport outside of inner boundary
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2) {
   C[i][j] = 1.0;
  }
 }// end loop through y values
}// end loop through x values
// end set up diffusion and initial probability distribution
   arrays


// normalize the probability distribution
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  pOff[i][j] = pOff[i][j] / norm;
 }
}


// start creating filaments that will make up the network

// "filament length" depends on physical filament lengths
// and the system's spatial resolution
int filArrayLength;
```

```
filArrayLength = (int)(l/dx);

// filament number index
int q = 0;

// position along filament index
int k;

// indices indicating where new piece of filament will be
   placed
int r, s;

// used in establishing filament locations
double radPos, angPos, theta, xPos, yPos;
double radPos2, xPos2, yPos2, diff;

// used to indicate filament polarity
double pol;

// indicate where filament "grows", once piece at a time
double xNew, yNew;

srand((unsigned)time(NULL));

// start setting up the filament network
while(q < numFil) {
 // initial radial postion of filament (inner endpoint)
 radPos = outer - (inner) * ((double)rand()/(double)RAND_MAX)
    ;
 // initial angular position of filament
 angPos = ((double)rand()/(double)RAND_MAX) * (2 * M_PI);
 // angle filament makes with respect to "radially outward"
 theta = (-M_PI) * ((double)rand()/(double)RAND_MAX) + (M_PI
    /2);
 // random polarity for the current filament
 // outward (1) or inward (-1)
 pol = (-2) * ((double)rand()/(double)RAND_MAX) + 1;
 pol = pol / fabs(pol);

 // positions of filament endpoints
 xPos = xc + radPos * cos(angPos);
 yPos = yc + radPos * sin(angPos);
 xPos2 = xPos + l * cos(angPos + theta);
 yPos2 = yPos + l * sin(angPos + theta);

 // radial position of outer endpoint
```

```
radPos2 = sqrt(pow((xPos2 - xc), 2) + pow((yPos2 - yc), 2));

// shift filaments in or out if any part of them is outside
   the cell
// transport area
if(radPos < (inner + 0.2)){
 diff = (inner + 0.2) - radPos;
 radPos = radPos + diff;
 radPos2 = radPos2 + diff;
}
if(radPos2 > outer){
 diff = radPos2 - outer;
 radPos = radPos - diff;
 radPos2 = radPos2 - diff;
}

// if the filament was moved, change the endpoint positions
xPos = xc + radPos * cos(angPos);
yPos = yc + radPos * sin(angPos);

xPos2 = xPos + l * cos(angPos + theta);
yPos2 = yPos + l * sin(angPos + theta);


// "grow" the filament
for(k = 0; k < filArrayLength + 1; k++){
 // new piece of the filament
 xNew = xPos + k * dx * cos(angPos + theta);
 yNew = yPos + k * dy * sin(angPos + theta);
 // convert to indices
 r = (int)(xNew/dx);
 s = (int)(yNew/dy);

 // start storing filament information into arrays
 // filaments will be "two indices" wide
 for(i = 0; i < 2; i++){
  for(j = 0; j < 2; j++){
   // make sure filament is "grown" inside the cell
   if(pow(((r+i)*dx-xc),2) + pow(((s+j)*dy-yc),2) < outer2){
    // indicate where the piece of the filament will be
       located
    filNet[r+i][s+j] = 1.0;
    // velocity along the filament
    vx[r+i][s+j] = vTot * pol * cos(angPos + theta);
    vy[r+i][s+j] = vTot * pol * sin(angPos + theta);
    pols[r+i][s+j] = pol;
```

```
      // set the filament attachment
      // and detachment rate
      kOn[r+i][s+j] = Kon;
      kOff[r+i][s+j] = Koff;
      // if pol = -1 set inner end as "endpoint"
      if(pol == -1.0 && k == 0){
       filEnds[r+i][s+j] = 1.0;

      }
      // if pol = 1 set outer end as "endpoint"
      if(pol == 1.0 && k == (filArrayLength)){
       filEnds[r+i][s+j] = 1.0;

      }
     }
    }
   }
  }// end growing a filament

  // get ready to lay down the next filament
  q++;

}// end laying down filament network

// evolve the probability distribution for "N" time steps
for (m = 0; m < N; m++) {

  // update distribution
  for (i = 1; i < NX - 1; i++) {
   for (j = 1; j < NY - 1; j++) {

    // allow on and off switching where a filament exists
    // before the distribution moves
    pOnNew[i][j] = dt * (kOn[i][j]*pOff[i][j] - kOff[i][j]*pOn
       [i][j]);
    pOffNew[i][j] = dt * (kOff[i][j]*pOn[i][j] - kOn[i][j]*
       pOff[i][j]);

    // diffusion off filaments
    // allow "off" distribution to move first
    pOffNew[i][j] =  pOffNew[i][j] +  dt*(
      D[i][j]*C[i][j]*(C[i+1][j]*(pOff[i+1][j]-pOff[i][j]) + C
         [i-1][j]*(pOff[i-1][j]-pOff[i][j]))/dx2
                  +D[i][j]*C[i][j]*(C[i][j+1]*(pOff[i][j+1]-
                     pOff[i][j]) + C[i][j-1]*(pOff[i][j-1]-
                     pOff[i][j]))/dy2);
```

```
// movement along filaments
pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                   ])/(dx)*(C[i+1][j])*filNet[i+1][j]
                +(0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                   -1][j])/(dx)*(C[i][j])*filNet[i][j]
              +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                 /(dx)*(C[i-1][j])*filNet[i-1][j]
            +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i
               +1][j])/(dx)*(C[i][j])*filNet[i][j]
              +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                 ])/(dy)*(C[i][j+1])*filNet[i][j+1]
            +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j
               -1])/(dy)*(C[i][j])*filNet[i][j]
              +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                 /(dy)*(C[i][j-1])*filNet[i][j-1]
            +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][
               j+1])/(dy)*(C[i][j])*filNet[i][j]);


// falling off a filament endpoint
pOffNew[i][j] = pOffNew[i][j] + dt * C[i][j] * (
                 (0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                    -1][j])/(dx)*(C[i][j])*filEnds[i-1][j]
        +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i+1][j
           ])/(dx)*(C[i][j])*filEnds[i+1][j]
        +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j-1])
           /(dy)*(C[i][j])*filEnds[i][j-1]
        +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][j
           +1])/(dy)*(C[i][j])*filEnds[i][j+1]);


// leaving a filament endpoint
pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                   ])/(dx)*(C[i+1][j])*filEnds[i][j]
              +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                 /(dx)*(C[i-1][j])*filEnds[i][j]
              +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                 ])/(dy)*(C[i][j+1])*filEnds[i][j]
              +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                 /(dy)*(C[i][j-1])*filEnds[i][j]);
```

```
  }// end looping through "y" indices
 }// end looping through "x" indices

 // final updates
 for (i = 1; i < NX - 1; i++) {
  for (j = 1; j < NY - 1; j++) {
   // apply changes to the off and on distributions that
   // were calculated above
   pOff[i][j] += pOffNew[i][j];
   pOn[i][j] += pOnNew[i][j];
   p[i][j] = pOff[i][j] + pOn[i][j];
   // if the distribution leaves the cell, eliminate it
   if(pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > outer2){
    p[i][j] = 0.0;
   }
   // calculate the survival probability at the current time
      step
   S[m] += p[i][j] * dx * dy;
  }
 }

 // check survival probability
 //printf("Survival probability: %lf\n", S[m]);

}// end evolving distribution

// calculate fptd and mfpt
for (m = 0; m < N; m++){
 t = m * dt;
 if(m == N - 1){
  F[m] = fabs(-(S[m] - S[m - 1]) / dt);
 } else if(m == 0){
  F[m] = fabs(-(S[m + 1] - S[m]) / dt);
 } else {
  F[m] = fabs(-(S[m + 1] - S[m - 1]) / (2 * dt));
 }
 mfpt += t * F[m] * dt;

 // check fptd
 //printf("FPTD: %lf\n", F[m]);

}

// check mfpt
```

```
 //printf("MFPT: %lf\n", mfpt);

 // output desired quantities to txt files
 FILE *outpProbs;
 FILE *outpPols;
 FILE *outpSurvival;
 FILE *outpFPTD;
 FILE *outpMFPT;

 outpProbs = fopen("probs.txt", "w");
 outpPols = fopen("pols.txt", "w");
 outpSurvival = fopen("survival.txt", "w");
 outpFPTD = fopen("fptd.txt", "w");
 outpMFPT = fopen("mfpt.txt", "w");


 for (i = 0; i < NX; i++) {
  for (j = 0; j < NY; j++) {
   fprintf(outpProbs, "%lf ", p[i][j]);
   fprintf(outpPols, "%lf ", pols[i][j]);
  }
  fprintf(outpProbs, "\n");
  fprintf(outpPols, "\n");
 }

 for (m = 0; m < N; m++){
  fprintf(outpSurvival, "%lf\n", S[m]);
  fprintf(outpFPTD, "%lf\n", F[m]);
 }

 fprintf(outpMFPT, "%lf\n", mfpt);

 fclose(outpProbs);
 fclose(outpPols);
 fclose(outpSurvival);
 fclose(outpFPTD);
 fclose(outpMFPT);

 return (0);
}
```

7.2.2 transNetNumVaryKs.c

Computes the time evolution of a cargo distribution for different values of k_{on} and k_{off}.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define N 10 // number of time steps
#define NX 210 // number of x coordinates
#define NY 210 // number of y coordinates

// global variables

// sets resolution and distance step size and max x and y
   values
double dx = 0.1, dy = 0.1, xmax, ymax, xc, yc, dx2, dy2;

// arrays to be used in integration of advection diffusion
   equation

// total probability distribution, "off" distribution, "on"
   distribution
double p[NX][NY], pOff[NX][NY], pOn[NX][NY];
double pOnNew[NX][NY], pOffNew[NX][NY];

// "off" and "on" switching rates
double kOff[NX][NY], kOn[NX][NY];

// diffusion constant
double D[NX][NY];

// velocity field arrays
double vx[NX][NY], vx2[NX][NY], vy[NX][NY], vy2[NX][NY];

// indicator for outside inner radius and inside outer radius
double C[NX][NY];

// array that indicates where the filaments are located
double filNet[NX][NY];

// array that indicates where filament endpoints are located
double filEnds[NX][NY];

// array of filament polarities
```

```
double pols[NX][NY];

// survival probability
double S[N];

// first passage time distribution
double F[N];

// mean first passage time
// initialize to 0.0
//double mfpt;

// number of filaments and filament length
int numFil = 150;
double l = 3.0;

// number of networks to integrate over
int numNets = 5;

// current network index
int netNum;

// size of time step
double dt = 0.01;

// how much time has passed
double t;

// radius of inner and outer boundaries
double outer = 10.0, inner = 5.0, outer2, inner2;

// initial (radial) width of the probability distribution
double width = 0.2;

// starting filament on and off rates and velocity along
   filaments
double Koff = 0.0, Kon = 0.0, vTot = 1.0;

// amount on and off rates change for each distribution
   evolution
double dKoff = 0.4, dKon = 4.0;

// max on and off rates reached in each distribution evolution
double KoffMax = 1.6, KonMax = 20.0;

// number of Koff and Kon values
```

```
int nKoff, nKon;

// Koff and Kon indices
int nOff, nOn;

// bulk diffusion constant
double Db = 0.051;

// strings used in file names
char sProbs[50];
char sPols[50];
char sSurvival[50];
char sFPTD[50];
char sMFPT[50];

char sEnd[50];
char sEnd2[50];

// output data to txt files
FILE *outpProbs;
FILE *outpPols;
FILE *outpSurvival;
FILE *outpFPTD;
FILE *outpMFPT;
FILE *outpKoff;
FILE *outpKon;

int main()
{

 // position (i, j)  and time indices (m) for arrays
 int i, j, m;

 // total time elapsed, time spend off filaments, time spent
    on filaments
 double t, tOff, tOn;

 // "hypotenuse" step size
 double dr, dr2;

 // probability distribution normalization constant
 double norm;


 // setting initial global variable values
```

```
// max (x, y) coordinates
xmax = NX * dx;
ymax = NY * dy;

// (x, y) coordinates of the cell's center
xc = xmax / 2;
yc = ymax / 2;

dx2 = dx * dx;
dy2 = dy * dy;
dr2 = dx2 + dy2;
dr = sqrt(dr2);

// squares of the inner and outer radii
inner2 = inner * inner;
outer2 = outer * outer;

// number of Koff and Kon values
nKoff = (int)(KoffMax / dKoff) + 1;
nKon = (int)(KonMax / dKon) + 1;

// array of mfpts (one for each koff/kon pair)
double mfpt[nKoff][nKon];

// seed random number
srand((unsigned)time(NULL));

// integrate over multiple networks
for(netNum = 0; netNum < numNets; netNum++){


// start creating filaments that will make up the network

// "filament length" depends on physical filament lengths
// and the system's spatial resolution
int filArrayLength;
filArrayLength = (int)(l/dx);

// filament number index
int q = 0;

// position along filament index
int k;

// indices indicating where new piece of filament will be
   placed
```

```
int r, s;

// used in establishing filament locations
double radPos, angPos, theta, xPos, yPos;
double radPos2, xPos2, yPos2, diff;

// used to indicate filament polarity
// radially outward (+1) or inward (-1)
double pol;

// indicate where filament "grows", once piece at a time
double xNew, yNew;

// clear the network arrays
for(i = 0; i < NX; i++){
 for(j = 0; j < NY; j++){
  vx[i][j] = 0.0;
  vy[i][j] = 0.0;
  kOn[i][j] = 0.0;
  kOff[i][j] = 0.0;
  filNet[i][j] = 0.0;
  filEnds[i][j] = 0.0;
  pols[i][j] = 0.0;
 }
}



// start setting up the filament network
while(q < numFil) {
 // initial radial postion of filament (inner endpoint)
 radPos = outer - (inner) * ((double)rand()/(double)RAND_MAX)
    ;
 // initial angular position of filament
 angPos = ((double)rand()/(double)RAND_MAX) * (2 * M_PI);
 // angle filament makes with respect to "radially outward"
 theta = (-M_PI) * ((double)rand()/(double)RAND_MAX) + (M_PI
    /2);
 // random polarity for the current filament
 // outward (+1) or inward (-1)
 pol = (-2) * ((double)rand()/(double)RAND_MAX) + 1;
 pol = pol / fabs(pol);

 // positions of filament endpoints

 // inner endpoint
```

```
xPos = xc + radPos * cos(angPos);
yPos = yc + radPos * sin(angPos);

// outer endpoint
xPos2 = xPos + l * cos(angPos + theta);
yPos2 = yPos + l * sin(angPos + theta);

// radial position of outer endpoint
radPos2 = sqrt(pow((xPos2 - xc), 2) + pow((yPos2 - yc), 2));

// shift filaments in or out if any part of them is outside
   the cell
// transport area
if(radPos < (inner + 0.2)){
 diff = (inner + 0.2) - radPos;
 radPos = radPos + diff;
 radPos2 = radPos2 + diff;
}
if(radPos2 > outer){
 diff = radPos2 - outer;
 radPos = radPos - diff;
 radPos2 = radPos2 - diff;
}

// if the filament was moved, change the endpoint positions
xPos = xc + radPos * cos(angPos);
yPos = yc + radPos * sin(angPos);

xPos2 = xPos + l * cos(angPos + theta);
yPos2 = yPos + l * sin(angPos + theta);


// "grow" the filament
for(k = 0; k < filArrayLength + 1; k++){
 // new piece of the filament
 xNew = xPos + k * dx * cos(angPos + theta);
 yNew = yPos + k * dy * sin(angPos + theta);
 // convert to indices
 r = (int)(xNew/dx);
 s = (int)(yNew/dy);

 // start storing filament information into arrays
 // filaments will be "two indices" wide
 for(i = 0; i < 2; i++){
  for(j = 0; j < 2; j++){
   // make sure filament is "grown" inside the cell
```

```
    if(pow(((r+i)*dx-xc),2) + pow(((s+j)*dy-yc),2) < outer2){
     // indicate where the piece of the filament will be
        located
     filNet[r+i][s+j] = 1.0;
     // velocity along the filament
     vx[r+i][s+j] = vTot * pol * cos(angPos + theta);
     vy[r+i][s+j] = vTot * pol * sin(angPos + theta);
     pols[r+i][s+j] = pol;

     // if pol = -1 set inner end as "filamnt end"
     if(pol == -1.0 && k == 0){
      filEnds[r+i][s+j] = 1.0;

     }
     // if pol = 1 set outer end as "filament end"
     if(pol == 1.0 && k == (filArrayLength)){
      filEnds[r+i][s+j] = 1.0;

     }
    }
   }
  }
 }// end growing a filament

 // get ready to lay down the next filament
 q++;

}// end laying down filament network

// start looping through different off and on rates

for(nOff = 0; nOff < nKoff; nOff++){
 Koff = nOff * dKoff;
for(nOn = 0; nOn < nKon; nOn++){
 Kon = nOn * dKon;

// probability distribution normalization factor
// initialize to 0.0
norm = 0.0;

// set up diffusion and initial probability distribution
   arrays
// also set the filament attachment and detachment rates
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  // diffusion constant is the same everywhere
```

```
  D[i][j] = Db;
  // probability distribution begins in a "ring", off all
     filaments
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2 &&
      pow((i*dx-xc), 2) + pow((j*dy-yc), 2) < pow((inner+
         width), 2)) {
   pOff[i][j] = 1.0;
  } else {
   pOff[i][j] = 0.0;
  }
  pOn[i][j] = 0.0;
  p[i][j] = 0.0;
  // keep track of normalization factor
  norm = norm + pOff[i][j]*dx*dy;
  // inner cell boundary is reflecting
  // only allow transport outside of inner boundary
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2) {
   C[i][j] = 1.0;
  } else {
   C[i][j] = 0.0;
  }
  // if a filament exists, set the attachment and detachment
     rate
  if (filNet[i][j] == 1.0) {
   kOff[i][j] = Koff;
   kOn[i][j] = Kon;
  } else {
   kOff[i][j] = 0.0;
   kOn[i][j] = 0.0;
  }
 }// end loop through y values
}// end loop through x values
// end set up diffusion and initial probability distribution
   arrays


// normalize the probability distribution
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  pOff[i][j] = pOff[i][j] / norm;
 }
}

// intialize fpt variables and arrays
mfpt[nOff][nOn] = 0.0;
```

```
for (m = 0; m < N; m++){
 S[m] = 0.0;
 F[m] = 0.0;
}

//check Koff and Kon values
//printf("kOff: %lf\tkOn: %lf\n", Koff, Kon);

// evolve the probability distribution for "N" time steps
for (m = 0; m < N; m++) {

 // update distribution
 for (i = 1; i < NX - 1; i++) {
  for (j = 1; j < NY - 1; j++) {

   // allow on and off switching where a filament exists
   // before the distribution moves
   pOnNew[i][j] = dt * (kOn[i][j]*pOff[i][j] - kOff[i][j]*pOn
      [i][j]);
   pOffNew[i][j] = dt * (kOff[i][j]*pOn[i][j] - kOn[i][j]*
      pOff[i][j]);

   // diffusion off filaments
   // allow "off" distribution to move first
   pOffNew[i][j] =  pOffNew[i][j] +  dt*(
             D[i][j]*C[i][j]*(C[i+1][j]*(pOff[i+1][j]-pOff[i
                ][j]) + C[i-1][j]*(pOff[i-1][j]-pOff[i][j]))/
                dx2
                  +D[i][j]*C[i][j]*(C[i][j+1]*(pOff[i][j+1]-
                     pOff[i][j]) + C[i][j-1]*(pOff[i][j-1]-
                     pOff[i][j]))/dy2);

   // movement along filaments
   pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                   (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                      ])/(dx)*(C[i+1][j])*filNet[i+1][j]
                   +(0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                      -1][j])/(dx)*(C[i][j])*filNet[i][j]
                  +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                     /(dx)*(C[i-1][j])*filNet[i-1][j]
                 +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i
                    +1][j])/(dx)*(C[i][j])*filNet[i][j]
                  +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                     ])/(dy)*(C[i][j+1])*filNet[i][j+1]
                 +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j
                    -1])/(dy)*(C[i][j])*filNet[i][j]
```

```
                  +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                     /(dy)*(C[i][j-1])*filNet[i][j-1]
                 +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][
                    j+1])/(dy)*(C[i][j])*filNet[i][j]);

  // falling off a filament endpoint (movement to "off" from
      "on")
  pOffNew[i][j] = pOffNew[i][j] + dt * C[i][j] * (
                   (0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                      -1][j])/(dx)*(C[i][j])*filEnds[i-1][j]
                 +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i
                    +1][j])/(dx)*(C[i][j])*filEnds[i+1][j]
                 +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j
                    -1])/(dy)*(C[i][j])*filEnds[i][j-1]
                 +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][
                    j+1])/(dy)*(C[i][j])*filEnds[i][j+1]);


  // leaving a filament endpoint (movement from "on" to "off
     ")
  pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                   (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                      ])/(dx)*(C[i+1][j])*filEnds[i][j]
                  +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                     /(dx)*(C[i-1][j])*filEnds[i][j]
                  +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                     ])/(dy)*(C[i][j+1])*filEnds[i][j]
                  +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                     /(dy)*(C[i][j-1])*filEnds[i][j]);

 }// end looping through "y" indices
}// end looping through "x" indices

// final probability updates
for (i = 1; i < NX - 1; i++) {
 for (j = 1; j < NY - 1; j++) {
  // apply changes to the off and on distributions that
  // were calculated above
  pOff[i][j] += pOffNew[i][j];
  pOn[i][j] += pOnNew[i][j];
  // calculate total probability distribution
  p[i][j] = pOff[i][j] + pOn[i][j];
  // if the distribution leaves the cell, eliminate it
  if(pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > outer2){
   p[i][j] = 0.0;
   pOff[i][j] = 0.0;
```

```
    pOn[i][j] = 0.0;
   }
   // calculate the survival probability at the current time
      step
   S[m] += p[i][j] * dx * dy;
  }
 }// end probability updates

 // check survival probability
 //printf("Survival probability: %lf\n", S[m]);

}// end evolving distribution

// calculate fptd and mfpt
for (m = 0; m < N; m++){
 t = m * dt;
 if(m == N - 1){
  F[m] = fabs(-(S[m] - S[m - 1]) / dt);
 } else if(m == 0){
  F[m] = fabs(-(S[m + 1] - S[m]) / dt);
 } else {
  F[m] = fabs(-(S[m + 1] - S[m - 1]) / (2 * dt));
 }
 mfpt[nOff][nOn] += t * F[m] * dt;

 // check fptd
 //printf("FPTD: %lf\n", F[m]);

}

// check mfpt
//printf("MFPT: %lf\n", mfpt);


// start outputting data to files

sprintf(sProbs, "probs");
sprintf(sSurvival, "survival");
sprintf(sFPTD, "fptd");

// end of each file name
// depends on the current network number and the
// current off and on rate
sprintf(sEnd, "netNum%dkOff%.2fkOn%.2f.txt", netNum, Koff,
   Kon);
```

```
outpProbs = fopen(strcat(sProbs,sEnd), "w");
outpSurvival = fopen(strcat(sSurvival,sEnd), "w");
outpFPTD = fopen(strcat(sFPTD,sEnd), "w");

// output remaining probability distribution
// for this koff/kon pair to a file
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  fprintf(outpProbs, "%lf ", p[i][j]);

 }
 fprintf(outpProbs, "\n");

}

// output survival probability and fptd
// for this koff/kon pair to a file
for (m = 0; m < N; m++){
 fprintf(outpSurvival, "%lf\n", S[m]);
 fprintf(outpFPTD, "%lf\n", F[m]);
}

// close files that have already been written to
fclose(outpProbs);
fclose(outpSurvival);
fclose(outpFPTD);

}// end looping through Kon values
}// end looping through Koff values

// prepare to output filament polarity
// and mfpt for each koff/kon pair
// for the current network number
sprintf(sPols, "polsVaryK");
sprintf(sMFPT, "mfptVaryK");

sprintf(sEnd2, "netNum%d.txt", netNum);


outpPols = fopen(strcat(sPols,sEnd2), "w");
outpMFPT = fopen(strcat(sMFPT,sEnd2), "w");


outpKoff = fopen("kOffs.txt", "w");
outpKon = fopen("kOns.txt", "w");
```

```
 // output mfpt, koff, and kon values to files
 for (nOff = 0; nOff < nKoff; nOff++){
  Koff = nOff * dKoff;
  for (nOn = 0; nOn < nKon; nOn++){
   Kon = nOn * dKon;
   fprintf(outpMFPT, "%lf ", mfpt[nOff][nOn]);
   fprintf(outpKoff, "%lf ", Koff);
   fprintf(outpKon, "%lf ", Kon);

  }
  fprintf(outpMFPT, "\n");
  fprintf(outpKoff, "\n");
  fprintf(outpKon, "\n");
 }

 // output filament polarities to a file
 for (i = 0; i < NX; i++){
  for (j = 0; j < NY; j++){
   fprintf(outpPols, "%lf ", pols[i][j]);

  }
  fprintf(outpPols, "\n");
 }

 // close remaining output files
 fclose(outpMFPT);
 fclose(outpPols);
 fclose(outpKoff);
 fclose(outpKon);


 }// end calculations for this network



 return (0);
}
```

7.2.3 transNetNumVaryPols.c

Computes the time evolution of a cargo distribution for different filament outward polarization probabilities.

```
#include <stdio.h>
#include <math.h>
```

```
#include <stdlib.h>
#include <time.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

#define N 10 // number of time steps
#define NX 210 // number of x coordinates
#define NY 210 // number of y coordinates

// global variables

// sets resolution and distance step size and max x and y
   values
double dx = 0.1, dy = 0.1, xmax, ymax, xc, yc, dx2, dy2;

// arrays to be used in integration of advection diffusion
   equation

// total probability distribution, "off" distribution, "on"
   distribution
double p[NX][NY], pOff[NX][NY], pOn[NX][NY];
double pOnNew[NX][NY], pOffNew[NX][NY];

// "off" and "on" switching rates
double kOff[NX][NY], kOn[NX][NY];

// diffusion constant
double D[NX][NY];

// velocity field arrays
double vx[NX][NY], vx2[NX][NY], vy[NX][NY], vy2[NX][NY];

// indicator for outside inner radius and inside outer radius
double C[NX][NY];

// array that indicates where the filaments are located
double filNet[NX][NY];

// array that indicates where filament endpoints are located
double filEnds[NX][NY];

// array of filament polarities
double pols[NX][NY];
```

```
// survival probability
double S[N];

// first passage time distribution
double F[N];

// mean first passage time
// initialize to 0.0
double mfpt;

// outward polarization bias
double polBias = 0.0;
double polBiasMax = 1.0;
double dP = 0.1;

// number of polarization biases used
int numPols;

// plolarization bias index
int nP;

// number of filaments and filament length
int numFil = 150;
double l = 1.0;
double lMax = 5.0;
double dL = 1.0;

// number of filament lengths used
int numLengths;

// filament length index
int nL;

// number of networks
int numNets = 5;

// network number index
int netNum;

// size of time step
double dt = 0.01;

// how much time has passed
double t;

// radius of inner and outer boundaries
```

```
double outer = 10.0, inner = 5.0, outer2, inner2;

// initial (radial) width of the probability distribution
double width = 0.2;

// starting filament on and off rates and velocity along
   filaments
double Koff = 1.0, Kon = 5.0, vTot = 1.0;

// bulk diffusion constant
double Db = 0.051;

// strings used in file names
char sProbs[50];
char sPols[50];
char sSurvival[50];
char sFPTD[50];
char sMFPT[50];

char sEnd[50];
char sEnd2[50];

// output data to txt files
FILE *outpProbs;
FILE *outpPols;
FILE *outpSurvival;
FILE *outpFPTD;
FILE *outpMFPT;
FILE *outpPB;
FILE *outpFL;

int main()
{

 // position (i, j)  and time indices (m) for arrays
 int i, j, m;

 // total time elapsed, time spend off filaments, time spent
    on filaments
 double t, tOff, tOn;

 // "hypotenuse" step size
 double dr, dr2;

 // probability distribution normalization constant
 double norm;
```

```
// setting initial global variable values

// max (x, y) coordinates
xmax = NX * dx;
ymax = NY * dy;

// (x, y) coordinates of the cell's center
xc = xmax / 2;
yc = ymax / 2;

dx2 = dx * dx;
dy2 = dy * dy;
dr2 = dx2 + dy2;
dr = sqrt(dr2);

// squares of the inner and outer radii
inner2 = inner * inner;
outer2 = outer * outer;

// number of Koff and Kon values
numPols = (int)(polBiasMax / dP) + 1;
numLengths = (int)(lMax / dL) + 1;

// array of mfpts (one for length/polarization bias pair)
double mfpt[numPols][numLengths];


// filament length by number of indices
int filArrayLength;

// filament number index
int q;

// position along filament index
int k;

// indices indicating where new piece of filament will be
   placed
int r, s;

// used in establishing filament locations
double radPos, angPos, theta, xPos, yPos;
double radPos2, xPos2, yPos2, diff;
```

```
// used to indicate filament polarity
// radially outward (+1) or inward (-1)
double pol;

// indicate where filament "grows", once piece at a time
double xNew, yNew;

// seed random number
srand((unsigned)time(NULL));

// loop over different network realizations
for(netNum = 0; netNum < numNets; netNum++){


// start looping through different plolarization biases
// and different filament lengths

for(nP = 0; nP < numPols; nP++){
 polBias = nP * dP;
for(nL = 1; nL < numLengths; nL++){
 l = nL * dL;

// initialize filament network arrays
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  filNet[i][j] = 0.0;
  filEnds[i][j] = 0.0;
  pols[i][j] = 0.0;
  vx[i][j] = 0.0;
  vy[i][j] = 0.0;
 }
}

// start creating filaments that will make up the network

// "filament length" depends on physical filament lengths
// and the system's spatial resolution
filArrayLength = (int)(l/dx);

q = 0;

// start setting up the filament network
while(q < numFil) {
 // initial radial postion of filament (inner endpoint)
 radPos = outer - (inner) * ((double)rand()/(double)RAND_MAX)
    ;
```

```
// initial angular position of filament
angPos = ((double)rand()/(double)RAND_MAX) * (2 * M_PI);
// angle filament makes with respect to "radially outward"
theta = (-M_PI) * ((double)rand()/(double)RAND_MAX) + (M_PI
   /2);
// random polarity for the current filament
// outward (+1) or inward (-1)
pol = (-1) * ((double)rand()/(double)RAND_MAX) + polBias *
   1;
pol = pol / fabs(pol);

// positions of filament endpoints

// inner endpoint
xPos = xc + radPos * cos(angPos);
yPos = yc + radPos * sin(angPos);

// outer endpoint
xPos2 = xPos + l * cos(angPos + theta);
yPos2 = yPos + l * sin(angPos + theta);

// radial position of outer endpoint
radPos2 = sqrt(pow((xPos2 - xc), 2) + pow((yPos2 - yc), 2));

// shift filaments in or out if any part of them is outside
   the cell
// transport area
if(radPos < (inner + 0.2)){
 diff = (inner + 0.2) - radPos;
 radPos = radPos + diff;
 radPos2 = radPos2 + diff;
}
if(radPos2 > outer){
 diff = radPos2 - outer;
 radPos = radPos - diff;
 radPos2 = radPos2 - diff;
}

// if the filament was moved, change the endpoint positions
xPos = xc + radPos * cos(angPos);
yPos = yc + radPos * sin(angPos);

xPos2 = xPos + l * cos(angPos + theta);
yPos2 = yPos + l * sin(angPos + theta);
```

```
 // "grow" the filament
 for(k = 0; k < filArrayLength + 1; k++){
  // new piece of the filament
  xNew = xPos + k * dx * cos(angPos + theta);
  yNew = yPos + k * dy * sin(angPos + theta);
  // convert to indices
  r = (int)(xNew/dx);
  s = (int)(yNew/dy);

  // start storing filament information into arrays
  // filaments will be "two indices" wide
  for(i = 0; i < 2; i++){
   for(j = 0; j < 2; j++){
    // make sure filament is "grown" inside the cell
    if(pow(((r+i)*dx-xc),2) + pow(((s+j)*dy-yc),2) < outer2){
     // indicate where the piece of the filament will be
        located
     filNet[r+i][s+j] = 1.0;
     // velocity along the filament
     vx[r+i][s+j] = vTot * pol * cos(angPos + theta);
     vy[r+i][s+j] = vTot * pol * sin(angPos + theta);
     pols[r+i][s+j] = pol;

     // if pol = -1 set inner end as "filamnt end"
     if(pol == -1.0 && k == 0){
      filEnds[r+i][s+j] = 1.0;

     }
     // if pol = 1 set outer end as "filament end"
     if(pol == 1.0 && k == (filArrayLength)){
      filEnds[r+i][s+j] = 1.0;

     }
    }
   }
  }
 }// end growing a filament

 // get ready to lay down the next filament
 q++;

}// end laying down filament network



// probability distribution normalization factor
```

```
// initialize to 0.0
norm = 0.0;

// set up diffusion and initial probability distribution
   arrays
// also set the filament attachment and detachment rates
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  // diffusion constant is the same everywhere
  D[i][j] = Db;
  // probability distribution begins in a "ring", off all
     filaments
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2 &&
      pow((i*dx-xc), 2) + pow((j*dy-yc), 2) < pow((inner+
         width), 2)) {
   pOff[i][j] = 1.0;
  } else {
   pOff[i][j] = 0.0;
  }
  pOn[i][j] = 0.0;
  p[i][j] = 0.0;
  // keep track of normalization factor
  norm = norm + pOff[i][j]*dx*dy;
  // inner cell boundary is reflecting
  // only allow transport outside of inner boundary
  if (pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > inner2) {
   C[i][j] = 1.0;
  } else {
   C[i][j] = 0.0;
  }
  // if a filament exists, set the attachment and detachment
     rate
  if (filNet[i][j] == 1.0) {
   kOff[i][j] = Koff;
   kOn[i][j] = Kon;
  } else {
   kOff[i][j] = 0.0;
   kOn[i][j] = 0.0;
  }
 }// end loop through y values
}// end loop through x values
// end set up diffusion and initial probability distribution
   arrays


// normalize the probability distribution
```

```
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  pOff[i][j] = pOff[i][j] / norm;
 }
}

// intialize fpt variables and arrays
mfpt[nP][nL] = 0.0;

for (m = 0; m < N; m++){
 S[m] = 0.0;
 F[m] = 0.0;
}

//check Koff and Kon values
//printf("kOff: %lf\tkOn: %lf\n", Koff, Kon);

// evolve the probability distribution for "N" time steps
for (m = 0; m < N; m++) {

 // update distribution
 for (i = 1; i < NX - 1; i++) {
  for (j = 1; j < NY - 1; j++) {

   // allow on and off switching where a filament exists
   // before the distribution moves
   pOnNew[i][j] = dt * (kOn[i][j]*pOff[i][j] - kOff[i][j]*pOn
      [i][j]);
   pOffNew[i][j] = dt * (kOff[i][j]*pOn[i][j] - kOn[i][j]*
      pOff[i][j]);

   // diffusion off filaments
   // allow "off" distribution to move first
   pOffNew[i][j] =  pOffNew[i][j] +  dt*(
             D[i][j]*C[i][j]*(C[i+1][j]*(pOff[i+1][j]-pOff[i
                ][j]) + C[i-1][j]*(pOff[i-1][j]-pOff[i][j]))/
                dx2
                  +D[i][j]*C[i][j]*(C[i][j+1]*(pOff[i][j+1]-
                     pOff[i][j]) + C[i][j-1]*(pOff[i][j-1]-
                     pOff[i][j]))/dy2);

   // movement along filaments
   pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                  (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                     ])/(dx)*(C[i+1][j])*filNet[i+1][j]
```

```
                    +(0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                       -1][j])/(dx)*(C[i][j])*filNet[i][j]
                   +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                      /(dx)*(C[i-1][j])*filNet[i-1][j]
                  +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i
                     +1][j])/(dx)*(C[i][j])*filNet[i][j]
                   +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                      ])/(dy)*(C[i][j+1])*filNet[i][j+1]
                  +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j
                     -1])/(dy)*(C[i][j])*filNet[i][j]
                   +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                      /(dy)*(C[i][j-1])*filNet[i][j-1]
                  +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][
                     j+1])/(dy)*(C[i][j])*filNet[i][j]);

  // falling off a filament endpoint (movement to "off" from
      "on")
  pOffNew[i][j] = pOffNew[i][j] + dt * C[i][j] * (
                    (0.5*(vx[i-1][j]+fabs(vx[i-1][j])))*(pOn[i
                       -1][j])/(dx)*(C[i][j])*filEnds[i-1][j]
                  +(0.5*(vx[i+1][j]-fabs(vx[i+1][j])))*(-pOn[i
                     +1][j])/(dx)*(C[i][j])*filEnds[i+1][j]
                  +(0.5*(vy[i][j-1]+fabs(vy[i][j-1])))*(pOn[i][j
                     -1])/(dy)*(C[i][j])*filEnds[i][j-1]
                  +(0.5*(vy[i][j+1]-fabs(vy[i][j+1])))*(-pOn[i][
                     j+1])/(dy)*(C[i][j])*filEnds[i][j+1]);


  // leaving a filament endpoint (movement from "on" to "off
     ")
  pOnNew[i][j] = pOnNew[i][j] + dt * C[i][j] * (
                   (0.5*(vx[i][j]+fabs(vx[i][j])))*(-pOn[i][j
                      ])/(dx)*(C[i+1][j])*filEnds[i][j]
                  +(0.5*(vx[i][j]-fabs(vx[i][j])))*(pOn[i][j])
                     /(dx)*(C[i-1][j])*filEnds[i][j]
                   +(0.5*(vy[i][j]+fabs(vy[i][j])))*(-pOn[i][j
                      ])/(dy)*(C[i][j+1])*filEnds[i][j]
                   +(0.5*(vy[i][j]-fabs(vy[i][j])))*(pOn[i][j])
                      /(dy)*(C[i][j-1])*filEnds[i][j]);

 }// end looping through "y" indices
}// end looping through "x" indices

// final probability updates
for (i = 1; i < NX - 1; i++) {
 for (j = 1; j < NY - 1; j++) {
```

```
    // apply changes to the off and on distributions that
    // were calculated above
    pOff[i][j] += pOffNew[i][j];
    pOn[i][j] += pOnNew[i][j];
    // calculate total probability distribution
    p[i][j] = pOff[i][j] + pOn[i][j];
    // if the distribution leaves the cell, eliminate it
    if(pow((i*dx-xc), 2) + pow((j*dy-yc), 2) > outer2){
     p[i][j] = 0.0;
     pOff[i][j] = 0.0;
     pOn[i][j] = 0.0;
    }
    // calculate the survival probability at the current time
       step
    S[m] += p[i][j] * dx * dy;
   }
  }// end probability updates

  // check survival probability
  //printf("Survival probability: %lf\n", S[m]);

 }// end evolving distribution

 // calculate fptd and mfpt
 for (m = 0; m < N; m++){
  t = m * dt;
  if(m == N - 1){
   F[m] = fabs(-(S[m] - S[m - 1]) / dt);
  } else if(m == 0){
   F[m] = fabs(-(S[m + 1] - S[m]) / dt);
  } else {
   F[m] = fabs(-(S[m + 1] - S[m - 1]) / (2 * dt));
  }
  mfpt[nP][nL] += t * F[m] * dt;

  // check fptd
  //printf("FPTD: %lf\n", F[m]);

 }

 // check mfpt
 //printf("MFPT: %lf\n", mfpt);


 // start outputting data to files
```

```
sprintf(sProbs, "probs");
sprintf(sSurvival, "survival");
sprintf(sFPTD, "fptd");

// prepare to output filament polarities
sprintf(sPols, "pols");

// end of each file name
// depends on current polarization bias
// and filament length
sprintf(sEnd, "netNum%dPB%.2fFL%.2f.txt", netNum, polBias, l)
   ;


outpProbs = fopen(strcat(sProbs,sEnd), "w");
outpPols = fopen(strcat(sPols,sEnd), "w");

outpSurvival = fopen(strcat(sSurvival,sEnd), "w");
outpFPTD = fopen(strcat(sFPTD,sEnd), "w");

// output remaining probability distribution
// and filament polarities
// for this polarization bias/filament length pair to a file
for (i = 0; i < NX; i++) {
 for (j = 0; j < NY; j++) {
  fprintf(outpProbs, "%lf ", p[i][j]);
  fprintf(outpPols, "%lf ", pols[i][j]);

 }
 fprintf(outpProbs, "\n");
 fprintf(outpPols, "\n");

}

// output survival probability and fptd
// for this polarization bias/filament length pair to a file
for (m = 0; m < N; m++){
 fprintf(outpSurvival, "%lf\n", S[m]);
 fprintf(outpFPTD, "%lf\n", F[m]);
}

// close files that have already been written to
fclose(outpProbs);
fclose(outpPols);
fclose(outpSurvival);
fclose(outpFPTD);
```

```
 }// end looping through filament lengths
 }// end looping through polarization biases

 // prepare to output mfpt for each polarization bias/filament
     length pair

 sprintf(sMFPT, "mfpt");

 sprintf(sEnd2, "netNum%d.txt", netNum);

 outpMFPT = fopen(strcat(sMFPT,sEnd2), "w");

 outpPB = fopen("polBiases.txt", "w");
 outpFL = fopen("filLengths.txt", "w");

 // output mfpt, polarization biases, and filament lengths
    values to files
 for (nP = 0; nP < numPols; nP++){
  polBias = nP * dP;
  for (nL = 1; nL < numLengths; nL++){
   l = nL * dL;
   fprintf(outpMFPT, "%lf ", mfpt[nP][nL]);
   fprintf(outpPB, "%lf ", polBias);
   fprintf(outpFL, "%lf ", l);

  }
  fprintf(outpMFPT, "\n");
  fprintf(outpPB, "\n");
  fprintf(outpFL, "\n");
 }


 // close remaining output files
 fclose(outpMFPT);
 fclose(outpPB);
 fclose(outpFL);
 }// end integrating over different network realizations

 return (0);
}
```

7.2.4 plotNums.py

Plots different filament configurations and states of the cargo distributions on the filament networks.

```
#import scipy as sp
import matplotlib.pyplot as plt
import matplotlib
#import math

# open file and return the contents
def openFile(fileName):
    with open(fileName) as file:
        result = [[float(digit) for digit in line.split()] for
            line in file]
    return result



def main():

    xmax = 21.0
    ymax = 21.0

    probDist1 = openFile('probsnetNum1kOff0.00kOn8.00.txt')
    pols1 = openFile('polsnetNum1.txt')
    probDist2 = openFile('probsnetNum2kOff0.00kOn8.00.txt')
    pols2 = openFile('polsnetNum2.txt')
    #survival = openFile('survivalPB0.00FL5.00.txt')
    #fptd = openFile('fptdPB0.00FL5.00.txt')


    # plot the contents of the file results
    fig = plt.figure(4)

    img1 = plt.subplot2grid((2,2),(0,0))
    im1 = img1.imshow(probDist1, cmap = matplotlib.cm.hot,
       origin = 'lower',\
                   extent = [0.0, xmax, 0.0, ymax], vmax =
                      0.01)
    img1.set_xlabel('x ($\mu$m)')
    img1.set_ylabel('y ($\mu$m)')
    fig.colorbar(im1,ax=img1).set_label('$P = P_{on} + P_{off}
       $')


    img2 = plt.subplot2grid((2,2),(0,1))
    im2 = img2.imshow(pols1, cmap = matplotlib.cm.seismic,
       origin = 'lower',\
                   extent = [0.0, xmax, 0.0, ymax])
    img2.set_xlabel('x ($\mu$m)')
```

```
img2.set_ylabel('y ($\mu$m)')
fig.colorbar(im2,ax=img2).set_label('Filament Polarization
    (in or out)')

img1 = plt.subplot2grid((2,2),(1,0))
im1 = img1.imshow(probDist2, cmap = matplotlib.cm.hot,
   origin = 'lower',\
               extent = [0.0, xmax, 0.0, ymax], vmax =
                  0.01)
img1.set_xlabel('x ($\mu$m)')
img1.set_ylabel('y ($\mu$m)')
fig.colorbar(im1,ax=img1).set_label('$P = P_{on} + P_{off}
   $')


img2 = plt.subplot2grid((2,2),(1,1))
im2 = img2.imshow(pols2, cmap = matplotlib.cm.seismic,
   origin = 'lower',\
               extent = [0.0, xmax, 0.0, ymax])
img2.set_xlabel('x ($\mu$m)')
img2.set_ylabel('y ($\mu$m)')
fig.colorbar(im2,ax=img2).set_label('Filament Polarization
    (in or out)')



#plt.subplot2grid((2,2),(1,0))
#plt.plot(survival)
#plt.xlabel('time steps (0.01 s each)')
#plt.ylabel('Survival probability')

#plt.subplot2grid((2,2),(1,1))
#plt.plot(fptd)
#plt.ylim(0,0.01)
#plt.xlabel('time steps (0.01 s each)')
#plt.ylabel('FPTD')




plt.tight_layout()
plt.show()
```

```
main()
```

7.2.5 randNumTester.py

Confirms the Python random number generator.

```
import random
import matplotlib.pyplot as plt

def main():

    randList = []
    randCounts = {'negative': 0, 'positive': 0}

    probPositive = 0.8

    for i in range(100):
        newRand = -1 * random.random() + probPositive
        newRand = newRand / abs(newRand)
        randList.append(newRand)

    for i in range(len(randList)):
        if randList[i] == -1.0:
            randCounts['negative'] += 1
        if randList[i] == 1.0:
            randCounts['positive'] += 1

    print randCounts

    #plt.plot(randCounts)
    #plt.ylim(0,100)
    #plt.show()



main()
```

7.2.6 plotSurvivalProbs.py

Determines survival probabilities for different networks.

```
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
```

```
# open file and return the contents
def openFile(fileName):
    with open(fileName) as file:
        result = [[float(digit) for digit in line.split()] for
            line in file]
    return result


def main():

    pb = 0.0
    fl = 1.0

    survival = []
    pbs = []
    fls = []



    i = 0
    while pb <= 1.0:
        survival.append([])
        pbs.append([])
        fls.append([])
        fl = 1.0
        while fl <= 5.0:
            survivalTemp = np.array(openFile('survivalPB'+str(
               pb)+'0'+'FL'+str(fl)+'0'+'.txt'))
            survivalTemp2 = survivalTemp[len(survivalTemp)
               -1,0]
            survival[i].append(survivalTemp2)
            pbs[i].append(pb)
            fls[i].append(fl)
            fl += 1.0
            #print kOff, kOn

        i += 1
        pb += 0.1

    survival = np.array(survival).transpose()
    mfpt = np.array(openFile('mfpt.txt')).transpose()


    fls = np.array(fls).transpose()
    pbs = np.array(pbs).transpose()
```

```
    #print survival2

    # plot the contents of the file results
    fig = plt.figure(1)

    #img = plt.subplot2grid((1,1),(0,0))
    #im = img.imshow(mfpt[:,7:], origin = 'lower',cmap = 'jet
       ',\
    #                  extent = [0.7, 1.0, 1.0, 5.0], aspect =
        0.1, interpolation = 'bilinear')

    #img.set_xlabel('Polarization bias')
    #img.set_ylabel('Filament length ($\mu$m)')
    #fig.colorbar(im,ax=img).set_label('MFPT (s)')

    # plot different 1d curves

    polBiases = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
       0.9, 1.0]

    plt.subplot2grid((1,1),(0,0))
    plt.plot(polBiases[:6], survival[0,:6],marker='o',label='
       Filament length = 1 $\mu$m')
    plt.plot(polBiases[:6], survival[1,:6],marker='o',label='
       Filament length = 2 $\mu$m')
    plt.plot(polBiases[:6], survival[2,:6],marker='o',label='
       Filament length = 3 $\mu$m')
    plt.plot(polBiases[:6], survival[3,:6],marker='o',label='
       Filament length = 4 $\mu$m')
    plt.plot(polBiases[:6], survival[4,:6],marker='o',label='
       Filament length = 5 $\mu$m')

    plt.legend(loc='upper right')
    plt.ylabel('Survival probability')
    plt.xlabel('Polarization bias')

    plt.tight_layout()
    plt.show()



main()
```

7.2.7 plotProbVariance.py

Calculates variation in survival probabilities over different networks.

```
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
import math

# open file and return the contents
def openFile(fileName):
    with open(fileName) as file:
        result = [[float(digit) for digit in line.split()] for
            line in file]
    return result


def main():

    kOffs = []
    kOns = []

    probAvgs = []
    probStdevs = []

    kOff = 0.0
    kOn = 0.0

    k = 0

    pb = 0.0
    fl = 1.0

    while pb <= 1.0:
        kOffs.append([])
        kOns.append([])
        # prepare a new empty list to hold different
           distribution data
        probAvgs.append([])
        probStdevs.append([])

        kOn = 0.0
        fl = 1.0
        while fl <= 5.0:

            kOffs[k].append(kOff)
            kOns[k].append(kOn)
```

```
            # read the probability distribution data at the
               final time step
            probDistTemp = openFile('probsPB'+str(pb)+'0'+'FL
               '+str(fl)+'0'+'.txt')

            count = 0
            probSum = 0.0
            var = 0.0

            for i in range(len(probDistTemp)):
                for j in range(len(probDistTemp[i])):
                    probSum += probDistTemp[i][j]
                    count += 1
            # end summing distribution values

            # calculate the mean distribution
            mean = probSum / count
            probAvgs[k].append(mean)

            for i in range(len(probDistTemp)):
                for j in range(len(probDistTemp[i])):
                    var += ((probDistTemp[i][j] - mean)*(
                       probDistTemp[i][j] - mean)) / count
            # end calculating distribution variance

            # calculate the distribution standard deviation
            stdev = math.sqrt(var)
            probStdevs[k].append(stdev)

            kOn += 4.0
            fl += 1.0
        # end while kOn

        k += 1
        kOff += 0.4
        pb += 0.1
    # end while kOff

    kOffs = np.array(kOffs).transpose()
    kOns = np.array(kOns).transpose()

    probAvgs = np.array(probAvgs).transpose()
    probStdevs = np.array(probStdevs).transpose()

    # plot the contents of the file results
    fig = plt.figure(1)
```

```
    #img = plt.subplot2grid((1,1),(0,0))
    #im = img.imshow(probStdevs[:,:6], origin = 'lower',cmap =
        'jet',\
    #                   extent = [0.0, 0.6, 1.0, 5.0], aspect =
        0.1, interpolation = 'bilinear')

    #img.set_xlabel('Polarization bias')
    #img.set_ylabel('Filament length ($\mu$m)')
    #fig.colorbar(im,ax=img).set_label('Distribution standard
       deviation')

    polBiases = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
       0.9, 1.0]

    plt.subplot2grid((1,1),(0,0))
    plt.plot(polBiases[:6], probStdevs[0,:6],marker='o',label
       ='Filament length = 1 $\mu$m')
    plt.plot(polBiases[:6], probStdevs[1,:6],marker='o',label
       ='Filament length = 2 $\mu$m')
    plt.plot(polBiases[:6], probStdevs[2,:6],marker='o',label
       ='Filament length = 3 $\mu$m')
    plt.plot(polBiases[:6], probStdevs[3,:6],marker='o',label
       ='Filament length = 4 $\mu$m')
    plt.plot(polBiases[:6], probStdevs[4,:6],marker='o',label
       ='Filament length = 5 $\mu$m')

    plt.legend(loc='upper right')
    plt.ylabel('Distribution standart deviation')
    plt.xlabel('Polarization bias')



    plt.tight_layout()
    plt.show()



main()
```

7.2.8 plotMFPT.py

Plots the MFPTs for different network configurations.

```
import matplotlib.pyplot as plt
```

```
import matplotlib
import numpy as np

# open file and return the contents
def openFile(fileName):
    with open(fileName) as file:
        result = [[float(digit) for digit in line.split()] for
            line in file]
    return result


def main():


    mfpt0 = np.array(openFile('mfptVaryKnetNum0.txt'))
    pols0 = np.array(openFile('polsVaryKnetNum0.txt'))
    mfpt1 = np.array(openFile('mfptVaryKnetNum1.txt'))
    pols1 = np.array(openFile('polsVaryKnetNum1.txt'))
    mfpt2 = np.array(openFile('mfptVaryKnetNum2.txt'))
    pols2 = np.array(openFile('polsVaryKnetNum2.txt'))
    mfpt3 = np.array(openFile('mfptVaryKnetNum3.txt'))
    pols3 = np.array(openFile('polsVaryKnetNum3.txt'))
    mfpt4 = np.array(openFile('mfptVaryKnetNum4.txt'))
    pols4 = np.array(openFile('polsVaryKnetNum4.txt'))

    mfpt0 = mfpt0.transpose()
    #pols0 = pols0.transpose()
    mfpt1 = mfpt1.transpose()
    #pols1 = pols1.transpose()
    mfpt2 = mfpt2.transpose()
    #pols2 = pols2.transpose()
    mfpt3 = mfpt3.transpose()
    #pols3 = pols3.transpose()
    mfpt4 = mfpt4.transpose()
    #pols4 = pols4.transpose()

    #print mfpt2

    # plot the contents of the file results
    fig = plt.figure(1)

    #img1 = plt.subplot2grid((1,1),(0,0))
    #im1 = img1.imshow(mfpt4[1:,:], origin = 'lower',cmap = '
       jet',\
    #                  extent = [0.0, 1.6, 4.0, 20.0], aspect
       = 0.1, interpolation = 'bilinear')
```

```
    #img1.set_xlabel('Off rate ($s^{-1}$)')
    #img1.set_ylabel('On rate ($s^{-1}$)')
    #fig.colorbar(im1,ax=img1,fraction=0.046).set_label('MFPT
       (s)')

    img2 = plt.subplot2grid((1,1),(0,0))
    im2 = img2.imshow(pols1, origin = 'lower',cmap =
       matplotlib.cm.seismic,\
                     extent = [0.0, 21.0, 0.0, 21.0])

    img2.set_xlabel('x ($\mu$m)')
    img2.set_ylabel('y ($\mu$m)')
    fig.colorbar(im2,ax=img2,fraction=0.046).set_label('
       Filament Polarization (in or out)')



    plt.tight_layout()
    plt.show()



main()
```

7.3 Anomalous Transport Programs

7.3.1 simTransMainMSD.c

This program is used to simulate anomalous transport and get data to perform MSD analysis.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define M_PI 3.14159265358979323846


int main()
{
 //cell radius
 double outer = 20.0, outer2;
 outer2 = pow(outer,2);
```

```
//radius of the nucleus
double inner = 5.0, inner2;
inner2 = pow(inner,2);
//max x and y values
//same coordinate system as probability evolution system
double xmax = (outer * 2) + 1, ymax = (outer * 2) + 1;
//center of cell
double xCellCent = xmax / 2, yCellCent = ymax / 2;
//cargo radius
double cRad = 0.1;
//size of time step (seconds)
double dt;
//time passed in seconds
double t, tOn, tOff;
//other parameters (distances in micrometers)
double D = 0.051;

//choose either regular or anomalous diffusion
int ANOM = 1, REG = 0;

//choose whether or not to model insulin
int INS = 0;

//add a filament switching probability
//0.0 means no switching will occur
double switchProb = 0.0;
double probOn, probOff;

//speed along filaments
double v = 1.0;

//distance step size (will vary)
double distStep = 0.1, distStep2;
distStep2 = pow(distStep,2);

//time step during normal diffusion
double dtReg;
dtReg = (distStep2) / (4 * D);

//loop indices
int i, j, k, m, currentm;
//on network, off network, stop the simulation
int ON, OFF, STOP, SWITCHED;

//seed rand()
srand(time(NULL));
```

```
double randNum;

//set current number of filaments and filament length
int minFils = 1500;
double minLength = 5.0;

int numFils = minFils;
double filLength = minLength;

//max number of filaments and max filament length
int maxFils = 1500;
double maxLength = 5.0;

int dFils = 500;
double dLength = 1.0;

//number of cargos and number of networks
int numCargs = 10000, numNets = 1;

double minx1x2, maxx1x2, miny1y2, maxy1y2;

double rc, theta, xc, yc, d1, d2, d;
double initial_x, initial_y, initial_t;
double r1, r2, alpha, p, x1, x2, y1, y2, diff;
double phi, beta;
double xcNew, ycNew, rcNew;

//on and off rates (constant for now)
double kOn = 5.0, kOff = 1.0;

double psi, a, b, gamma;
//if normal diffusion, gamma = 1
if (REG == 1){
 gamma = 1;
}
else{
 gamma = 0.8;
}

//used in calculating MSD
int timeInt, timeIntMax = 20000;
double sd;
double msdArray[timeIntMax][3];

//used to calculate variations in MSD at 10s and 100s
```

```
double msdCurrent10, msdCurrent100;
double msd10[numNets], msd100[numNets];
double msdSum10, msdSum100;
double av10, av100, var10, var100;
double stdev10, stdev100;

//used for calculating average time spent on the network
double fracTimeOn[numCargs*numNets];
int currentCargo = 0;


char sEnd1[500];

//start looping over different filament lengths and numbers
while(filLength <= maxLength){
   numFils = minFils;
   while(numFils <= maxFils){

//redeclare filament network arrays
double filNet[numFils][4];
double filEnds[numFils][4];

for(i = 0; i < timeIntMax; i++){
 msdArray[i][0] = 0.0;
 msdArray[i][1] = 0.0;
 msdArray[i][2] = 0.0;

}

//initialize fracTimeOn back to 0.0
for(i = 0; i < numCargs*numNets; i++){
 fracTimeOn[i] = 0.0;
}
currentCargo = 0;

//for keeping track of msd variation at 10s and 100s
for(i = 0; i < numNets; i++){
 msd10[i] = 0.0;
 msd100[i] = 0.0;
}



//set up end of file name
```

```
sprintf(sEnd1,"kOn%.2fkOff%.2fnumFil%dfilLen%.2fnumNets%
   dnumCargs%dgamma%.2fINS%d.txt",kOn,kOff,numFils,filLength,
   numNets,numCargs,gamma,INS);

//start laying down different networks
for(int currentNet = 0; currentNet < numNets; currentNet +=
   1){

//set up the current network
for(j = 0; j < numFils; j += 1){
 //random radial starting position
 r1 = outer - (outer - inner)* (double)rand() / RAND_MAX;
 //random angular starting position
 theta = (2*M_PI) * (double)rand()/RAND_MAX;
 //alpha between -pi/2 and +pi/2
 alpha = -(M_PI) * (double)rand()/RAND_MAX + (M_PI/2);
 //random filament polarity
 //positive is "out" negative is "in"
 p = (-2) * (double)rand()/RAND_MAX + 1;
 //p is +1 or -1
 p = p / fabs(p);
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //make sure filament ends are within desired region
 //shift filament out
 if(r1 < (inner + 0.2)){
  diff = (inner+0.2)-r1;
  r1 = r1 + diff;
  r2 = r2 + diff;
 }
 //shift filament in
 if(r2 > outer){
  diff = r2 - outer;
  r1 = r1 - diff;
  r2 = r2 - diff;
 }
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
```

```
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //store filament endpoints
 filEnds[j][0] = x1; filEnds[j][1] = x2; filEnds[j][2] = y1;
    filEnds[j][3] = y2;
 //store values in filament array
 filNet[j][0] = r1; filNet[j][1] = theta; filNet[j][2] =
    alpha; filNet[j][3] = p;
}//end set up the network



for(int currentCarg = 0; currentCarg < numCargs; currentCarg
   += 1){
 //set time to zero
 t = 0.0;
 tOn = 0.0;
 tOff = 0.0;
 if (INS == 0){
  //starting radial position of cargo
  rc = (inner + 0.2) - (0.2)*((double)rand())/RAND_MAX;
 }
 //if we're modling insuling, cargos have a different
    starting distribution
 if (INS == 1){
  //this is assumin outer = 10.0 and inner = 5.0
  rc = 10 - 5 * sqrt(4 - ((double)rand()/RAND_MAX + 3));
 }


 //starting angular position of cargo
 beta = (2*M_PI)*((double)rand())/RAND_MAX;

 //starting x, y values of cargo
 xc = xCellCent + rc * cos(beta);
 yc = yCellCent + rc * sin(beta);

 //keep track of starting x, y values
 initial_x = xc;
 initial_y = yc;
 initial_t = t;

 //cargo start   s off the network
 OFF = 1;
 ON = 0;
 //the simulation has not stopped yet
```

```
STOP = 0;
//start letting cargo "walk"
while(STOP == 0){
 //allow on/off switching if possible
 if(OFF == 1 && ON == 0){
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
  }

  if (ANOM == 1){
   //anomalous diffusion
   randNum = (double)rand()/RAND_MAX;
   dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
  }


  //check for nearby filaments and
  //filament endpoints. must be within
  //cargo radius
  m = 0;
  while (m < numFils && ON != 1){
   d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m][1]-
      filEnds[m][0])*yc
      +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
         ][0])/
       sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
          filEnds[m][1]-filEnds[m][0]),2));

   d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
      ][2]),2));
   d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
      ][3]),2));

   minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs(
      filEnds[m][0]-filEnds[m][1]);
   maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs(
      filEnds[m][0]-filEnds[m][1]);
   miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs(
      filEnds[m][2]-filEnds[m][3]);
   maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs(
      filEnds[m][2]-filEnds[m][3]);

   //cargo is near a filament
   if(d1 < cRad || d2 < cRad ||
```

```
  (xc > minx1x2 && xc < maxx1x2 && yc > miny1y2 && yc <
     maxy1y2 &&
                                d < cRad)){

   //probability of switching on the network
   probOn = (double)rand()/RAND_MAX;
   if(probOn <= (kOn*dt)){
    ON = 1;
    theta = filNet[m][1];
    alpha = filNet[m][2];
    p = filNet[m][3];
    x1 = filEnds[m][0];
    x2 = filEnds[m][1];
    y1 = filEnds[m][2];
    y2 = filEnds[m][3];
    //current filament number
    currentm = m;
   }
  }
  m += 1;
 }
}

//if on, allow possibility of falling off or switching to a
//nearby filament
if(ON == 1 && OFF == 0){
 dt = distStep / v;
 probOff = (double)rand()/RAND_MAX;
 minx1x2 = 0.5 * fabs(x1+x2) - 0.5 * fabs(x1-x2);
 maxx1x2 = 0.5 * fabs(x1+x2) + 0.5 * fabs(x1-x2);
 miny1y2 = 0.5 * fabs(y1+y2) - 0.5 * fabs(y1-y2);
 maxy1y2 = 0.5 * fabs(y1+y2) + 0.5 * fabs(y1-y2);
 //cargo will fall of the current filament
 if(probOff <= (kOff*dt) ||
   (xc<minx1x2 || xc>maxx1x2 || yc<miny1y2 || yc>maxy1y2)){
  //cargo has fallen off the network
  ON = 0;
  OFF = 1;
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
  }

  if (ANOM == 1){
   //anomalous diffusion
   randNum = (double)rand()/RAND_MAX;
```

```
  dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
 }
}//end check if  cargo has fallen off

//cargo still on. Check for nearby filaments
if (ON == 1 && OFF == 0){
 //cycle through filament number (m)
 m = 0;
 //cargo has not switched yet
 SWITCHED = 0;
 //if there is a filament nearby, allow switching
 while (m < numFils && SWITCHED != 1){
  d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m
     ][1]-filEnds[m][0])*yc
     +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
        ][0])/
      sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
         filEnds[m][1]-filEnds[m][0]),2));

  d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
     ][2]),2));
  d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
     ][3]),2));

  minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs
     (filEnds[m][0]-filEnds[m][1]);
  maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs
     (filEnds[m][0]-filEnds[m][1]);
  miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs
     (filEnds[m][2]-filEnds[m][3]);
  maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs
     (filEnds[m][2]-filEnds[m][3]);

  if((d1 < cRad || d2 < cRad ||
    (xc > minx1x2 && xc < maxx1x2 && yc > miny1y2 && yc <
       maxy1y2 &&
                                          d < cRad)) && (
                                             currentm != m)
                                             ){

   //probability of switching to another filament
   probOn = (double)rand()/RAND_MAX;
   if(probOn <= (switchProb)){
    //cargo switches over to another filament
    SWITCHED = 1;
    theta = filNet[m][1];
```

```
      alpha = filNet[m][2];
      p = filNet[m][3];
      x1 = filEnds[m][0];
      x2 = filEnds[m][1];
      y1 = filEnds[m][2];
      y2 = filEnds[m][3];
      //update current filamnet number (m)
      currentm = m;
     }
    }
    m += 1;
   }
  }
}
//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 = distStep / v;
}
//random walk off filament
if(OFF == 1){
 //pick a random direction
 phi = (2*M_PI)*(double)rand()/RAND_MAX;
 //move in that directino
 xcNew = xc + (distStep) * cos(phi);
 ycNew = yc + (distStep) * sin(phi);
 tOff = tOff + dt;
}
//ballistic motion on filament
if(ON == 1){
 //move along filament
 xcNew = xc + p * distStep * cos(theta + alpha);
 ycNew = yc + p * distStep * sin(theta + alpha);
 tOn = tOn + dt;
}
//new radial position of cargo
rcNew = sqrt(pow((xcNew-xCellCent),2)+pow((ycNew-yCellCent)
   ,2));
//new angular position of cargo
beta = atan((ycNew-yCellCent)/(xcNew-xCellCent));
//check to see if cargo is inside nucleus
if(rcNew < inner){
```

```
 //if cargo is inside the nucleus, move it back out
 //to original position
 xcNew = xc;
 ycNew = yc;
}
//check to see if the cargo has left the cell
if(rcNew > outer){

 //for msd calculations, two reflecting
 //boundaries
 xcNew = xc;
 ycNew = yc;

}
//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 = t + dt;

//used in calculating msd
sd = pow((xc - initial_x),2) + pow((yc - initial_y),2);

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

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


if((t / dtReg) < (double)timeIntMax){
 timeInt = (int)(t / dtReg);
 msdArray[timeInt][0] += sd;
 msdArray[timeInt][1] += 1.0;
 msdArray[timeInt][2] = (double)timeInt;
}

if((t / dtReg) > (double)timeIntMax){
 STOP = 1;
}


}//end movement of current cargo
```

```
 fracTimeOn[currentCargo] = tOn / t;
 currentCargo += 1;

 msd10[currentNet] += msdCurrent10;
 msd100[currentNet] += msdCurrent100;

}//end movement of ALL cargos

//msd for all cargos at 10s and 100s
msd10[currentNet] = msd10[currentNet] / numCargs;
msd100[currentNet] = msd100[currentNet] / numCargs;



}//end laying down all networks

msdSum10 = 0.0; msdSum100 = 0.0;
var10 = 0.0; var100 = 0.0;

for(i = 0; i < numNets; i++){
 msdSum10 += msd10[i];
 msdSum100 += msd100[i];
}

av10 = msdSum10 / numNets; av100 = msdSum100/ numNets;

for(i = 0; i < numNets; i++){
 var10 += pow((msd10[i] - av10), 2) / numNets;
 var100 += pow((msd100[i] - av100), 2) / numNets;
}

stdev10 = sqrt(var10);
stdev100 = sqrt(var100);

FILE *outp10;
FILE *outp100;

char sBeg10[500] = "msdSTDEV10";
char sBeg100[500] = "msdSTDEV100";

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);

 FILE *outp;
 char sBeg[500] = "MSD";
 outp = fopen(strcat(sBeg,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);

 FILE *outpFracs;
 char sBegFracs[500] = "fracTimeOnMSD";
 outpFracs = fopen(strcat(sBegFracs,sEnd1),"w");
 for(i = 0; i < numCargs*numNets; i++){
  fprintf(outpFracs,"%lf\n",fracTimeOn[i]);
 }
 fclose(outpFracs);



    //increase number of filaments
    numFils += dFils;
    }//end looping through number of filaments

 //increase filament length
 filLength += dLength;
 }//end looping through filament lengths

 return 0;
}//end main
```

7.3.2 simTransMainTAMSD.c

This program is used to simulate anomalous transport and get data to perform TA-MSD analysis.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define M_PI 3.14159265358979323846
```

```
int main()
{
 //cell radius
 double outer = 10.0, outer2;
 outer2 = pow(outer,2);
 //radius of the nucleus
 double inner = 5.0, inner2;
 inner2 = pow(inner,2);
 //max x and y values
 //same coordinate system as probability evolution system
 double xmax = (outer * 2) + 1, ymax = (outer * 2) + 1;
 //center of cell
 double xCellCent = xmax / 2, yCellCent = ymax / 2;
 //cargo radius
 double cRad = 0.1;
 //size of time step (seconds)
 double dt;
 //time passed in seconds
 double t, tOn, tOff;
 //other parameters (distances in micrometers)
 double D = 0.051;

 //choose either regular or anomalous diffusion
 int ANOM = 1, REG = 0;

 //choose whether or not to model insulin
 int INS = 0;

 //add a filament switching probability
 //0.0 means no switching will occur
 double switchProb = 0.0;
 double probOn, probOff;

 //speed along filaments
 double v = 1.0;

 //distance step size (will vary)
 double distStep = 0.1, distStep2;
 distStep2 = pow(distStep,2);

 //time step during normal diffusion
 double dtReg;
 dtReg = (distStep2) / (4 * D);
```

```
//loop indices
int i, j, k, m, currentm;
//on network, off network, stop the simulation
int ON, OFF, STOP, SWITCHED;

//seed rand()
srand(time(NULL));

double randNum;

//set current number of filaments and filament length
int minFils = 500;
double minLength = 5.0;

int numFils = minFils;
double filLength = minLength;

//max number of filaments and max filament length
int maxFils = 500;
double maxLength = 5.0;

int dFils = 500;
double dLength = 1.0;

//number of cargos and number of networks
int numCargs = 2, numNets = 2;

double minx1x2, maxx1x2, miny1y2, maxy1y2;

double rc, theta, xc, yc, d1, d2, d;
double initial_x, initial_y, initial_t;
double r1, r2, alpha, p, x1, x2, y1, y2, diff;
double phi, beta;
double xcNew, ycNew, rcNew;

//on and off rates (constant for now)
double kOn = 5.0, kOff = 1.0;

double psi, a, b, gamma;
//if normal diffusion, gamma = 1
if (REG == 1){
 gamma = 1;
}
else{
 gamma = 0.8;
}
```

```
//used to calculate TA iMSD
int timeInt, timeIntMax = 20000;
int maxTime = timeIntMax * dtReg;
double xytArray[maxTime][3];
double timeAvgMsdArray[maxTime][numCargs*numNets];
double timeAvgMsdFixedS1[maxTime][numCargs*numNets];
double timeAvgMsdFixedS2[maxTime][numCargs*numNets];
double timeAvgMsdFixedS3[maxTime][numCargs*numNets];
int currentCargo = 0;
int passNum, counts;
double dis2tot, dis2, dx, dy;

//used to calculate average time spent on the network
double fracTimeOn[numCargs*numNets];

char sEnd1[500];

//start looping over different filament lengths and numbers
while(filLength <= maxLength){
   numFils = minFils;
   while(numFils <= maxFils){

//redeclare filament network arrays
double filNet[numFils][4];
double filEnds[numFils][4];

//initialize fracTimeOn back to 0.0
for(i = 0; i < numCargs*numNets; i++){
 fracTimeOn[i] = 0.0;
}


//initializations for TA MSD calculation
currentCargo = 0;


//set up end of file name
sprintf(sEnd1,"kOn%.2fkOff%.2fnumFil%dfilLen%.2fnumNets%
   dnumCargs%dgamma%.2fINS%d.txt",kOn,kOff,numFils,filLength,
   numNets,numCargs,gamma,INS);

//start laying down different networks
for(int currentNet = 0; currentNet < numNets; currentNet +=
   1){
```

```
//set up the current network
for(j = 0; j < numFils; j += 1){
 //random radial starting position
 r1 = outer - (outer - inner)* (double)rand() / RAND_MAX;
 //random angular starting position
 theta = (2*M_PI) * (double)rand()/RAND_MAX;
 //alpha between -pi/2 and +pi/2
 alpha = -(M_PI) * (double)rand()/RAND_MAX + (M_PI/2);
 //random filament polarity
 //positive is "out" negative is "in"
 p = (-2) * (double)rand()/RAND_MAX + 1;
 //p is +1 or -1
 p = p / fabs(p);
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //make sure filament ends are within desired region
 //shift filament out
 if(r1 < (inner + 0.2)){
  diff = (inner+0.2)-r1;
  r1 = r1 + diff;
  r2 = r2 + diff;
 }
 //shift filament in
 if(r2 > outer){
  diff = r2 - outer;
  r1 = r1 - diff;
  r2 = r2 - diff;
 }
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //store filament endpoints
 filEnds[j][0] = x1; filEnds[j][1] = x2; filEnds[j][2] = y1;
    filEnds[j][3] = y2;
 //store values in filament array
 filNet[j][0] = r1; filNet[j][1] = theta; filNet[j][2] =
    alpha; filNet[j][3] = p;
```

```
}//end set up the network



for(int currentCarg = 0; currentCarg < numCargs; currentCarg
   += 1){
 //set time to zero
 t = 0.0;
 tOn = 0.0;
 tOff = 0.0;
 if (INS == 0){
  //starting radial position of cargo
  rc = (inner + 0.2) - (0.2)*((double)rand())/RAND_MAX;
 }
 //if we're modling insuling, cargos have a different
    starting distribution
 if (INS == 1){
  //this is assumin outer = 10.0 and inner = 5.0
  rc = 10 - 5 * sqrt(4 - ((double)rand()/RAND_MAX + 3));
 }


 //starting angular position of cargo
 beta = (2*M_PI)*((double)rand())/RAND_MAX;

 //starting x, y values of cargo
 xc = xCellCent + rc * cos(beta);
 yc = yCellCent + rc * sin(beta);

 //keep track of starting x, y values
 initial_x = xc;
 initial_y = yc;
 initial_t = t;

 //initialize xyt array
 for(int temp_int = 0; temp_int< maxTime; temp_int += 1){
  xytArray[temp_int][0] = 0.0;
  xytArray[temp_int][1] = 0.0;
  xytArray[temp_int][2] = 0.0;
 }

 //initialize first xyt values
 xytArray[0][0] = initial_x;
 xytArray[0][1] = initial_y;
 xytArray[0][2] = initial_t;
```

```
//cargo start   s off the network
OFF = 1;
ON = 0;
//the simulation has not stopped yet
STOP = 0;
//start letting cargo "walk"
while(STOP == 0){
 //allow on/off switching if possible
 if(OFF == 1 && ON == 0){
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
  }

  if (ANOM == 1){
   //anomalous diffusion
   randNum = (double)rand()/RAND_MAX;
   dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
  }


  //check for nearby filaments and
  //filament endpoints. must be within
  //cargo radius
  m = 0;
  while (m < numFils && ON != 1){
   d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m][1]-
      filEnds[m][0])*yc
      +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
         ][0])/
       sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
          filEnds[m][1]-filEnds[m][0]),2));

   d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
      ][2]),2));
   d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
      ][3]),2));

   minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs(
      filEnds[m][0]-filEnds[m][1]);
   maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs(
      filEnds[m][0]-filEnds[m][1]);
   miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs(
      filEnds[m][2]-filEnds[m][3]);
```

```
  maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs(
     filEnds[m][2]-filEnds[m][3]);

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

   //probability of switching on the network
   probOn = (double)rand()/RAND_MAX;
   if(probOn <= (kOn*dt)){
    ON = 1;
    theta = filNet[m][1];
    alpha = filNet[m][2];
    p = filNet[m][3];
    x1 = filEnds[m][0];
    x2 = filEnds[m][1];
    y1 = filEnds[m][2];
    y2 = filEnds[m][3];
    //current filament number
    currentm = m;
   }
  }
  m += 1;
 }
}

//if on, allow possibility of falling off or switching to a
//nearby filament
if(ON == 1 && OFF == 0){
 dt = distStep / v;
 probOff = (double)rand()/RAND_MAX;
 minx1x2 = 0.5 * fabs(x1+x2) - 0.5 * fabs(x1-x2);
 maxx1x2 = 0.5 * fabs(x1+x2) + 0.5 * fabs(x1-x2);
 miny1y2 = 0.5 * fabs(y1+y2) - 0.5 * fabs(y1-y2);
 maxy1y2 = 0.5 * fabs(y1+y2) + 0.5 * fabs(y1-y2);
 //cargo will fall of the current filament
 if(probOff <= (kOff*dt) ||
   (xc<minx1x2 || xc>maxx1x2 || yc<miny1y2 || yc>maxy1y2)){
  //cargo has fallen off the network
  ON = 0;
  OFF = 1;
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
```

```
 }

 if (ANOM == 1){
  //anomalous diffusion
  randNum = (double)rand()/RAND_MAX;
  dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
 }
}//end check if  cargo has fallen off

//cargo still on. Check for nearby filaments
if (ON == 1 && OFF == 0){
 //cycle through filament number (m)
 m = 0;
 //cargo has not switched yet
 SWITCHED = 0;
 //if there is a filament nearby, allow switching
 while (m < numFils && SWITCHED != 1){
  d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m
     ][1]-filEnds[m][0])*yc
     +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
        ][0])/
      sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
         filEnds[m][1]-filEnds[m][0]),2));

  d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
     ][2]),2));
  d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
     ][3]),2));

  minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs
     (filEnds[m][0]-filEnds[m][1]);
  maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs
     (filEnds[m][0]-filEnds[m][1]);
  miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs
     (filEnds[m][2]-filEnds[m][3]);
  maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs
     (filEnds[m][2]-filEnds[m][3]);

  if((d1 < cRad || d2 < cRad ||
    (xc > minx1x2 && xc < maxx1x2 && yc > miny1y2 && yc <
       maxy1y2 &&
                                          d < cRad)) && (
                                             currentm != m)
                                             ){

   //probability of switching to another filament
```

```
    probOn = (double)rand()/RAND_MAX;
    if(probOn <= (switchProb)){
     //cargo switches over to another filament
     SWITCHED = 1;
     theta = filNet[m][1];
     alpha = filNet[m][2];
     p = filNet[m][3];
     x1 = filEnds[m][0];
     x2 = filEnds[m][1];
     y1 = filEnds[m][2];
     y2 = filEnds[m][3];
     //update current filamnet number (m)
     currentm = m;
    }
   }
   m += 1;
  }
 }
}
//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 = distStep / v;
}
//random walk off filament
if(OFF == 1){
 //pick a random direction
 phi = (2*M_PI)*(double)rand()/RAND_MAX;
 //move in that directino
 xcNew = xc + (distStep) * cos(phi);
 ycNew = yc + (distStep) * sin(phi);
 tOff = tOff + dt;
}
//ballistic motion on filament
if(ON == 1){
 //move along filament
 xcNew = xc + p * distStep * cos(theta + alpha);
 ycNew = yc + p * distStep * sin(theta + alpha);
 tOn = tOn + dt;
}
//new radial position of cargo
```

```
 rcNew = sqrt(pow((xcNew-xCellCent),2)+pow((ycNew-yCellCent)
    ,2));
 //new angular position of cargo
 beta = atan((ycNew-yCellCent)/(xcNew-xCellCent));
 //check to see if cargo is inside nucleus
 if(rcNew < inner){
  //if cargo is inside the nucleus, move it back out
  //to original position
  xcNew = xc;
  ycNew = yc;
 }
 //check to see if the cargo has left the cell
 if(rcNew > outer){

  //for msd calculations, two reflecting
  //boundaries
  xcNew = xc;
  ycNew = yc;

 }
 //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 = t + dt;


 if((t / dtReg) > (double)timeIntMax){
  STOP = 1;
 }

 if((int)t < maxTime && xytArray[(int)t][0] == 0.0){
  xytArray[(int)t][0] = xc;
  xytArray[(int)t][1] = yc;
  xytArray[(int)t][2] = t;
 }


}//end movement of current cargo

fracTimeOn[currentCargo] = tOn / t;

//testing xyt values
//if cargo doesn't move after a certain amount of time
//it stays where it is.
```

```
 for(int temp_int = 0; temp_int < maxTime; temp_int += 1){
  if(xytArray[temp_int][0] == 0.0){
   xytArray[temp_int][0] = xytArray[temp_int-1][0];
   xytArray[temp_int][1] = xytArray[temp_int-1][1];
   xytArray[temp_int][2] = xytArray[temp_int-1][2]+1;
  }
 }

//calculate TA MSD of a single trajectory and store it in an
   array for
//all cargos (for all networks)
passNum = 1;
while(passNum < maxTime){
 dis2tot = 0.0;
 counts = 0;
 i = passNum;
 while(i < maxTime){
  dx = xytArray[i][0] - xytArray[i-passNum][0];
  dy = xytArray[i][1] - xytArray[i-passNum][1];
  dis2 = dx*dx + dy*dy;
  dis2tot = dis2tot + dis2;
  counts += 1;
  i += 1;
 }//end incrementing i
 timeAvgMsdArray[passNum][currentCargo] = dis2tot / counts;
 passNum += 1;
}//end incrementing passNum

//calculation for TA MSD (fixed s)
//s = 1
passNum = 1;
while(passNum < maxTime){
 dis2tot = 0.0;
 counts = 0;
 i = 0;
 while(i < passNum){
  dx = xytArray[i+1][0] - xytArray[i][0];
  dy = xytArray[i+1][1] - xytArray[i][1];
  dis2 = dx*dx + dy*dy;
  dis2tot = dis2tot + dis2;
  counts += 1;
  i += 1;
 }//end incrementing i
 timeAvgMsdFixedS1[passNum][currentCargo] = dis2tot / counts;
 passNum += 1;
}//end incrementing passNum
```

```
//s=2
passNum = 2;
while(passNum < maxTime){
 dis2tot = 0.0;
 counts = 0;
 i = 0;
 while(i < passNum-1){
  dx = xytArray[i+2][0] - xytArray[i][0];
  dy = xytArray[i+2][1] - xytArray[i][1];
  dis2 = dx*dx + dy*dy;
  dis2tot = dis2tot + dis2;
  counts += 1;
  i += 1;
 }//end incrementing i
 timeAvgMsdFixedS2[passNum][currentCargo] = dis2tot / counts;
 passNum += 1;
}//end incrementing passNumi

//s=3
passNum = 3;
while(passNum < maxTime){
 dis2tot = 0.0;
 counts = 0;
 i = 0;
 while(i < passNum-2){
  dx = xytArray[i+3][0] - xytArray[i][0];
  dy = xytArray[i+3][1] - xytArray[i][1];
  dis2 = dx*dx + dy*dy;
  dis2tot = dis2tot + dis2;
  counts += 1;
  i += 1;
 }//end incrementing i
 timeAvgMsdFixedS3[passNum][currentCargo] = dis2tot / counts;
 passNum += 1;
}//end incrementing passNum

currentCargo += 1;

}//end movement of ALL cargos

}//end laying down all networks

//outputting TA MSDs to files
FILE *outp, *outp1, *outp2, *outp3;
char sBeg[500] = "TAMSD";
```

```
 char sBeg1[500] = "TAMSDs1";
 char sBeg2[500] = "TAMSDs2";
 char sBeg3[500] = "TAMSDs3";
 outp = fopen(strcat(sBeg,sEnd1),"w");
 outp1 = fopen(strcat(sBeg1,sEnd1),"w");
 outp2 = fopen(strcat(sBeg2,sEnd1),"w");
 outp3 = fopen(strcat(sBeg3,sEnd1),"w");
 for(i = 0; i < maxTime; i++){
  for(k = 0; k < numCargs*numNets; k++){
   fprintf(outp,"%lf\t",timeAvgMsdArray[i][k]);
   fprintf(outp1,"%lf\t",timeAvgMsdFixedS1[i][k]);
   fprintf(outp2,"%lf\t",timeAvgMsdFixedS2[i][k]);
   fprintf(outp3,"%lf\t",timeAvgMsdFixedS3[i][k]);
  }
  fprintf(outp,"\n");
  fprintf(outp1,"\n");
  fprintf(outp2,"\n");
  fprintf(outp3,"\n");
 }
 fclose(outp);
 fclose(outp1);
 fclose(outp2);
 fclose(outp3);


 FILE *outpFracs;
 char sBegFracs[500] = "fracTimeOnTAMSD";
 outpFracs = fopen(strcat(sBegFracs,sEnd1),"w");
 for(i = 0; i < numCargs*numNets; i++){
  fprintf(outpFracs,"%lf\n",fracTimeOn[i]);
 }
 fclose(outpFracs);


    //increase number of filaments
    numFils += dFils;
    }//end looping through number of filaments

 //increase filament length
 filLength += dLength;
 }//end looping through filament lengths

 return 0;
}//end main
```

7.3.3 simTransMainFPTD.c

This program is used to simulate anomalous transport and get data to perform FPTD analysis.

```
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define M_PI 3.14159265358979323846


#include <string.h>
int main()
{
 //cell radius
 double outer = 10.0, outer2;
 outer2 = pow(outer,2);
 //radius of the nucleus
 double inner = 5.0, inner2;
 inner2 = pow(inner,2);
 //max x and y values
 //same coordinate system as probability evolution system
 double xmax = (outer * 2) + 1, ymax = (outer * 2) + 1;
 //center of cell
 double xCellCent = xmax / 2, yCellCent = ymax / 2;
 //cargo radius
 double cRad = 0.1;
 //size of time step (seconds)
 double dt;
 //time passed in seconds
 double t, tOn, tOff;
 //other parameters (distances in micrometers)
 double D = 0.051;

 //choose either regular or anomalous diffusion
 int ANOM = 1, REG = 0;

 //choose whether or not to model insulin
 int INS = 1;

 //add a filament switching probability
 //0.0 means no switching will occur
 double switchProb = 0.0;
 double probOn, probOff;
```

```
//speed along filaments
double v = 1.0;

//distance step size (will vary)
double distStep = 0.1, distStep2;
distStep2 = pow(distStep,2);

//time step during normal diffusion
double dtReg;
dtReg = (distStep2) / (4 * D);

//loop indices
int i, j, k, m, currentm, index;
//on network, off network, stop the simulation
int ON, OFF, STOP, SWITCHED;

//seed rand()
srand(time(NULL));

double randNum;

//set current number of filaments and filament length
int minFils = 100;
double minLength = 1.0;

int numFils = minFils;
double filLength = minLength;

//max number of filaments and max filament length
int maxFils = 500;
double maxLength = 5.0;

int dFils = 100;
double dLength = 1.0;

//number of cargos and number of networks
int numCargs = 400, numNets = 100;

double minx1x2, maxx1x2, miny1y2, maxy1y2;

double rc, theta, xc, yc, d1, d2, d;
double initial_x, initial_y, initial_t;
double r1, r2, alpha, p, x1, x2, y1, y2, diff;
double phi, beta;
double xcNew, ycNew, rcNew;
```

```
//on and off rates (constant for now)
double kOn = 5.0, kOff = 1.0;

double psi, a, b, gamma;
//if normal diffusion, gamma = 1
if (REG == 1){
 gamma = 1;
}
else{
 gamma = 0.8;
}

//FOR CALCULATING FPTD
int maxSteps = 1000000, binSize = 1;
 double FPTD[maxSteps];
int stepNum;
int count10 = 0, count100 = 0;
int fluxOut10[numNets], fluxOut100[numNets];
double fluxSum10 = 0.0, fluxSum100 = 0.0;
double av10, av100, var10 = 0.0, var100 = 0.0, stdev10,
   stdev100;

//used in MFPT calculations
double cargoFPTs[numCargs];
double fptSum, fptVar;
double cargoMFPTs[numNets], cargoFPTstdev[numNets];
double mfptSum, mfptVar, stdevSum;
double totMFPT, totStdev, avgStdev;


//for calculating fraction of the time spent on the network
double fracTimeOn[numCargs*numNets];
int currentCargo = 0;

char sEnd1[500];

//start looping over different filament lengths and numbers
while(filLength <= maxLength){
   numFils = minFils;
   while(numFils <= maxFils){

//redeclare filament network arrays
double filNet[numFils][4];
double filEnds[numFils][4];

//initialize FPTD back to 0.0
```

```
for(int index = 0; index < maxSteps; index += 1){
 FPTD[index] = 0.0;
}

//initialize fracTimeOn back to 0.0
for(int index = 0; index < numCargs*numNets; index += 1){
 fracTimeOn[index] = 0.0;
}

currentCargo = 0;

for(i = 0; i < numNets; i++){
 fluxOut10[i] = 0;
 fluxOut100[i] = 0;
}

//set up end of file name
sprintf(sEnd1,"kOn%.2fkOff%.2fnumFil%dfilLen%.2fnumNets%
   dnumCargs%dgamma%.2fINS%d.txt",kOn,kOff,numFils,filLength,
   numNets,numCargs,gamma,INS);

//start laying down different networks
for(int currentNet = 0; currentNet < numNets; currentNet +=
   1){

count10 = 0;
count100 = 0;

//set up the current network
for(j = 0; j < numFils; j += 1){
 //random radial starting position
 r1 = outer - (outer - inner)* (double)rand() / RAND_MAX;
 //random angular starting position
 theta = (2*M_PI) * (double)rand()/RAND_MAX;
 //alpha between -pi/2 and +pi/2
 alpha = -(M_PI) * (double)rand()/RAND_MAX + (M_PI/2);
 //random filament polarity
 //positive is "out" negative is "in"
 p = (-2) * (double)rand()/RAND_MAX + 1;
 //p is +1 or -1
 p = p / fabs(p);
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
```

```
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //make sure filament ends are within desired region
 //shift filament out
 if(r1 < (inner + 0.2)){
  diff = (inner+0.2)-r1;
  r1 = r1 + diff;
  r2 = r2 + diff;
 }
 //shift filament in
 if(r2 > outer){
  diff = r2 - outer;
  r1 = r1 - diff;
  r2 = r2 - diff;
 }
 //x and y values of filament endpoints
 x1 = xCellCent + r1 * cos(theta); y1 = yCellCent + r1 * sin(
    theta);
 x2 = x1 + filLength*cos(theta+alpha); y2 = y1 + filLength*
    sin(theta+alpha);
 //"outer" end of filament
 r2 = sqrt(pow((x2-xCellCent),2)+pow((y2-yCellCent),2));
 //store filament endpoints
 filEnds[j][0] = x1; filEnds[j][1] = x2; filEnds[j][2] = y1;
    filEnds[j][3] = y2;
 //store values in filament array
 filNet[j][0] = r1; filNet[j][1] = theta; filNet[j][2] =
    alpha; filNet[j][3] = p;
}//end set up the network


//empty cargo fpt array
for(i = 0; i < numCargs; i++){
 cargoFPTs[i] = 0.0;
}



for(int currentCarg = 0; currentCarg < numCargs; currentCarg
   += 1){
 //set time to zero
 stepNum = 0;
 t = 0.0;
 tOn = 0.0;
 tOff = 0.0;
```

```
if (INS == 0){
 //starting radial position of cargo
 rc = (inner + 0.2) - (0.2)*((double)rand())/RAND_MAX;
}
//if we're modling insuling, cargos have a different
   starting distribution
if (INS == 1){
 //this is assumin outer = 10.0 and inner = 5.0
 rc = 10 - 5 * sqrt(4 - ((double)rand()/RAND_MAX + 3));
}


//starting angular position of cargo
beta = (2*M_PI)*((double)rand())/RAND_MAX;

//starting x, y values of cargo
xc = xCellCent + rc * cos(beta);
yc = yCellCent + rc * sin(beta);

//keep track of starting x, y values
initial_x = xc;
initial_y = yc;
initial_t = t;

//cargo start   s off the network
OFF = 1;
ON = 0;
//the simulation has not stopped yet
STOP = 0;
//start letting cargo "walk"
while(STOP == 0){
 //allow on/off switching if possible
 if(OFF == 1 && ON == 0){
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
  }

  if (ANOM == 1){
   //anomalous diffusion
   randNum = (double)rand()/RAND_MAX;
   dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
  }


  //check for nearby filaments and
```

```
//filament endpoints. must be within
//cargo radius
m = 0;
while (m < numFils && ON != 1){
 d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m][1]-
    filEnds[m][0])*yc
    +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
       ][0])/
     sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
        filEnds[m][1]-filEnds[m][0]),2));

 d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
    ][2]),2));
 d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
    ][3]),2));

 minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs(
    filEnds[m][0]-filEnds[m][1]);
 maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs(
    filEnds[m][0]-filEnds[m][1]);
 miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs(
    filEnds[m][2]-filEnds[m][3]);
 maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs(
    filEnds[m][2]-filEnds[m][3]);

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

  //probability of switching on the network
  probOn = (double)rand()/RAND_MAX;
  if(probOn <= (kOn*dt)){
   ON = 1;
   theta = filNet[m][1];
   alpha = filNet[m][2];
   p = filNet[m][3];
   x1 = filEnds[m][0];
   x2 = filEnds[m][1];
   y1 = filEnds[m][2];
   y2 = filEnds[m][3];
   //current filament number
   currentm = m;
  }
 }
```

```
  m += 1;
 }
}

//if on, allow possibility of falling off or switching to a
//nearby filament
if(ON == 1 && OFF == 0){
 dt = distStep / v;
 probOff = (double)rand()/RAND_MAX;
 minx1x2 = 0.5 * fabs(x1+x2) - 0.5 * fabs(x1-x2);
 maxx1x2 = 0.5 * fabs(x1+x2) + 0.5 * fabs(x1-x2);
 miny1y2 = 0.5 * fabs(y1+y2) - 0.5 * fabs(y1-y2);
 maxy1y2 = 0.5 * fabs(y1+y2) + 0.5 * fabs(y1-y2);
 //cargo will fall of the current filament
 if(probOff <= (kOff*dt) ||
   (xc<minx1x2 || xc>maxx1x2 || yc<miny1y2 || yc>maxy1y2)){
  //cargo has fallen off the network
  ON = 0;
  OFF = 1;
  if (REG == 1){
   //normal diffusion
   dt = dtReg;
  }

  if (ANOM == 1){
   //anomalous diffusion
   randNum = (double)rand()/RAND_MAX;
   dt = pow((-randNum+1),(-1/gamma))/(1/dtReg);
  }
 }//end check if  cargo has fallen off

 //cargo still on. Check for nearby filaments
 if (ON == 1 && OFF == 0){
  //cycle through filament number (m)
  m = 0;
  //cargo has not switched yet
  SWITCHED = 0;
  //if there is a filament nearby, allow switching
  while (m < numFils && SWITCHED != 1){
   d = fabs((filEnds[m][3]-filEnds[m][2])*xc-(filEnds[m
      ][1]-filEnds[m][0])*yc
      +filEnds[m][1]*filEnds[m][2]-filEnds[m][3]*filEnds[m
         ][0])/
       sqrt(pow((filEnds[m][3]-filEnds[m][2]),2)+pow((
          filEnds[m][1]-filEnds[m][0]),2));
```

```
    d1 = sqrt(pow((xc-filEnds[m][0]),2)+pow((yc-filEnds[m
       ][2]),2));
    d2 = sqrt(pow((xc-filEnds[m][1]),2)+pow((yc-filEnds[m
       ][3]),2));

    minx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])-0.5*fabs
       (filEnds[m][0]-filEnds[m][1]);
    maxx1x2 = 0.5*fabs(filEnds[m][0]+filEnds[m][1])+0.5*fabs
       (filEnds[m][0]-filEnds[m][1]);
    miny1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])-0.5*fabs
       (filEnds[m][2]-filEnds[m][3]);
    maxy1y2 = 0.5*fabs(filEnds[m][2]+filEnds[m][3])+0.5*fabs
       (filEnds[m][2]-filEnds[m][3]);

    if((d1 < cRad || d2 < cRad ||
      (xc > minx1x2 && xc < maxx1x2 && yc > miny1y2 && yc <
         maxy1y2 &&
                                          d < cRad)) && (
                                             currentm != m)
                                             ){

     //probability of switching to another filament
     probOn = (double)rand()/RAND_MAX;
     if(probOn <= (switchProb)){
      //cargo switches over to another filament
      SWITCHED = 1;
      theta = filNet[m][1];
      alpha = filNet[m][2];
      p = filNet[m][3];
      x1 = filEnds[m][0];
      x2 = filEnds[m][1];
      y1 = filEnds[m][2];
      y2 = filEnds[m][3];
      //update current filamnet number (m)
      currentm = m;
     }
    }
    m += 1;
   }
  }
}
//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 = distStep / v;
}
//random walk off filament
if(OFF == 1){
 //pick a random direction
 phi = (2*M_PI)*(double)rand()/RAND_MAX;
 //move in that directino
 xcNew = xc + (distStep) * cos(phi);
 ycNew = yc + (distStep) * sin(phi);
 tOff = tOff + dt;
}
//ballistic motion on filament
if(ON == 1){
 //move along filament
 xcNew = xc + p * distStep * cos(theta + alpha);
 ycNew = yc + p * distStep * sin(theta + alpha);
 tOn = tOn + dt;
}
//new radial position of cargo
rcNew = sqrt(pow((xcNew-xCellCent),2)+pow((ycNew-yCellCent)
   ,2));
//new angular position of cargo
beta = atan((ycNew-yCellCent)/(xcNew-xCellCent));
//check to see if cargo is inside nucleus
if(rcNew < inner){
 //if cargo is inside the nucleus, move it back out
 //to original position
 xcNew = xc;
 ycNew = yc;
}
//check to see if the cargo has left the cell
if(rcNew > outer){

 //if cargo has left the cell, stop cargo movement
 STOP = 1;

}
//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 = t + dt;
```

```
 }//end movement of current cargo

 fracTimeOn[currentCargo] = tOn / t;

    //add times to the last FPTD element if necessary
    if (t >= maxSteps){
       t = maxSteps - 1;
    }

    //a cargo escaped at time t
    stepNum = (int) t / binSize;
    FPTD[stepNum] = FPTD[stepNum] + 1;

    if(t <= 10.0){
       count10++;
    }

    if(t <= 100){
       count100++;
    }

 currentCargo += 1;

 //current time is a fpt
 cargoFPTs[currentCarg] = t;

}//end movement of ALL cargos


fluxOut10[currentNet] = count10;
fluxOut100[currentNet] = count100;

//calculate MFPT for all cargos on this network
fptSum = 0.0;
for(i = 0; i < numCargs; i++){
 fptSum += cargoFPTs[i];
}

//MFPT for this network
cargoMFPTs[currentNet] = fptSum / numCargs;

//calculate standard deviation of fpts
fptVar = 0.0;
for(i = 0; i < numCargs; i++){
 fptVar += pow((cargoFPTs[i] - cargoMFPTs[currentNet]),2) /
    numCargs;
```

```
}

//standard deviation of fpts for this network
cargoFPTstdev[currentNet] = sqrt(fptVar);


//printf("flux out 10: %d flux out 100: %d\n",count10,
   count100);

}//end laying down all networks

//calculate overall MFPT
mfptSum = 0.0;
for(i = 0; i < numNets; i++){
 mfptSum += cargoMFPTs[i];
}
totMFPT = mfptSum / numNets;


//calculate standard deviation of MFPTs for each network
mfptVar = 0.0;
for(i = 0; i < numNets; i++){
 mfptVar += pow((cargoMFPTs[i] - totMFPT),2) / numNets;
}
totStdev = sqrt(mfptVar);

//calculate the average standard deviations for cargos on a
//single network
stdevSum = 0.0;
for(i = 0; i < numNets; i++){
 stdevSum += cargoFPTstdev[i];
}
avgStdev = stdevSum / numNets;


fluxSum10 = 0.0; fluxSum100 = 0.0;
var10 = 0.0; var100 = 0.0;
for(index = 0; index < numNets; index++){
 fluxSum10 += fluxOut10[index];
 fluxSum100 += fluxOut100[index];
}


av10 = fluxSum10 / numNets; av100 = fluxSum100 / numNets;

for(index = 0; index < numNets; index++){
```

```
 var10 += pow((fluxOut10[index] - av10),2) / numNets;
 var100 += pow((fluxOut100[index] - av100),2) / numNets;
}

stdev10 = sqrt(var10);
stdev100 = sqrt(var100);

//printf("av10: %lf stdev10: %lf\n",av10,stdev10);
//printf("av100: %lf stdev100: %lf\n",av100,stdev100);

//output overall MFPT, MFPT standard deviation, and average
   standard
//deviation to a file
FILE *outpMFPT;
char sBegMFPT[500] = "infoMFPT";
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);


FILE *outp10;
FILE *outp100;

char sBeg10[500] = "out10";
char sBeg100[500] = "out100";

outp10 = fopen(strcat(sBeg10,sEnd1),"w");
outp100 = fopen(strcat(sBeg100,sEnd1),"w");

fprintf(outp10, "%lf\t%lf\n", av10, stdev10);
fprintf(outp100, "%lf\t%lf\n", av100, stdev100);

fclose(outp10);
fclose(outp100);


//output FPTD to a file
FILE *outp;
char sBeg[500] = "FPTD";
outp = fopen(strcat(sBeg,sEnd1),"w");
       for(i=0;i<maxSteps;i++){
   fprintf(outp,"%lf\n",FPTD[i]);
}
fclose(outp);
```

```
 FILE *outpFracs;
 char sBegFracs[500] = "fracTimeOnFPTD";
 outpFracs = fopen(strcat(sBegFracs,sEnd1),"w");
 for(i = 0; i < numCargs*numNets; i++){
    fprintf(outpFracs,"%lf\n",fracTimeOn[i]);
 }
 fclose(outpFracs);

    //increase number of filaments
    numFils += dFils;
    }//end looping through number of filaments

 //increase filament length
 filLength += dLength;
 }//end looping through filament lengths

 return 0;
}//end main
```

7.3.4 msdAnalysis.py

This program analyzes MSD data.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 06 15:15:28 2017

@author: GPLP
"""
import matplotlib.pyplot as plt
import matplotlib
import diffusion_analysis
import numpy as np
from scipy.optimize import curve_fit

def fitFunc(t,a,b):
    return a*t**(b)

#def fitFunc2(t,a,b):
#    return a*np.exp(b*t)


def main():
```

```
#with open('MSDkOn5.00kOff1.00
   numFil0filLen1numNets1numCargs10000gamma0.80INS0.txt')
   as file:
#with open('MSDkOn5.00kOff1.00
   numFil100filLen1numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil300filLen1numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil500filLen1numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil100filLen3numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil300filLen3numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil500filLen3numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil100filLen5numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil300filLen5numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
#with open('MSDkOn5.00kOff1.00
   numFil500filLen5numNets1numCargs10000gamma0.80INS0.txt
   ') as file:

fracTimeOnTemp = []
with open('fracTimeOnkOn5.00kOff1.00
   numFil500filLen1numNets1numCargs100gamma0.80INS0.txt')
   as fp:
    for line in fp:
        fracTimeOnTemp.append([float(line)])

with open('MSDkOn5.00kOff1.00
   numFil500filLen1numNets1numCargs10000gamma0.80INS0.txt
   ') as file:
    msdArray = [[float(digit) for digit in line.split()]
       for line in file]
    msdList = []
    msdList.append(0.0)
    for i in range(len(msdArray)):
```

```
        if i != 0:
            msdList.append(msdArray[i][0]/msdArray[i][1])


msdSupers = [[0.8391,0.7561,1.0986,1.0975,1.1740],\
             [1.3483,1.2523,1.4904,1.5763,1.6502],\
             [1.5787,1.7582,1.7849,1.5310,1.8622],\
             [1.8967,1.8230,1.9542,1.8414,1.8647],\
             [1.8046,1.8134,1.7298,1.7581,1.7633]]
msdSubs = [[0.7402,0.7562,0.7294,0.8523,0.7636],\
           [0.6929,0.7822,0.7932,0.7410,0.6467],\
           [0.5220,0.6956,0.6575,0.7497,0.5722],\
           [0.5410,0.5892,0.5389,0.5261,0.5206],\
           [0.4834,0.4456,0.4290,0.4366,0.4109]]
fracTimeOn = [[0.0071,0.0097,0.0145,0.0177,0.0266],\
              [0.0099,0.0178,0.0347,0.0482,0.0719],\
              [0.0182,0.0397,0.0550,0.0593,0.1358],\
              [0.0316,0.0891,0.0787,0.1143,0.1584],\
              [0.0782,0.1739,0.2083,0.2666,0.3121]]

dtReg = (0.1*0.1) / (4*0.051)
time_values = np.arange(0,2000*dtReg,dtReg)
msd_values = np.array(msdList)
t = np.linspace(0,100,100-0)

divideTime = 1 / dtReg

fitParams,other = curve_fit(fitFunc,time_values[:
   divideTime],msd_values[:divideTime])
fitParams2, other2 = curve_fit(fitFunc,time_values[
   divideTime:],msd_values[divideTime:])
print "superdiffusive exponent", fitParams[1]
print "subdiffusive exponent", fitParams2[1]




#print 20/dtReg

#plt.ylim(0,50)
#plt.xlim(0.1,100)
#plt.xlabel('time (s)')
#plt.ylabel('MSD ($\mu$m$^2$)')
plt.plot(time_values,msd_values,color='red')
plt.plot(time_values,fitFunc(time_values,fitParams[0],
   fitParams[1]))
```

```
plt.plot(time_values,fitFunc(time_values,fitParams2[0],
   fitParams2[1]))
plt.xscale('log')
plt.yscale('log')


#x = dtReg * np.arange(time_values)

#print dtReg

#print len(msdList)
#time_values = np.arange(len(msdList))
time_values = np.arange(0,2000*dtReg,dtReg)
msd_values = np.array(msdList)
results = diffusion_analysis.fit_anomalous_diffusion_data(
   time_values,msd_values)
D,D_std,alpha,alpha_std=results[0:4]

x_vals,y_vals=results[4:]
#plt.plot(time_values,msd_values,color='green')
#plt.plot(x_vals,y_vals,color='blue')
#print alpha, alpha_std



fracTimeOnList = []
for i in range(len(fracTimeOnTemp)):
    fracTimeOnList.append(fracTimeOnTemp[i][0])

avgFracTimeOn = sum(fracTimeOnList) / len(fracTimeOnList)
print "average fraction of time on network: ",
   avgFracTimeOn



#color maps
#fig = plt.figure()
#im = plt.imshow(fracTimeOn, origin = 'lower',extent =
   [100,500,1,5],aspect = 100)
#im = plt.imshow(msdSupers, origin = 'lower',extent =
   [100,500,1,5],aspect = 100)
#im = plt.imshow(msdSubs, origin = 'lower',extent =
   [100,500,1,5],aspect = 100)
#fig.colorbar(im)
```

```
main()
```

7.3.5 msdAnalysis2.py

This program analyzes additional MSD data.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Mar 06 15:15:28 2017

@author: GPLP
"""
import matplotlib.pyplot as plt
import matplotlib
import numpy as np
from scipy.optimize import curve_fit

def fitFunc(t,a,b):
    return a*t**(b)

#def fitFunc2(t,a,b):
#    return a*np.exp(b*t)


def toList(fileName):
    with open(fileName) as file:
        msdArray = [[float(digit) for digit in line.split()]
           for line in file]
        fileList = []
        fileList.append(0.0)
        for i in range(1,len(msdArray)):
            if msdArray[i][1]>0.0:
                fileList.append(msdArray[i][0]/msdArray[i][1])
            else:
                fileList.append(0.0)
    return fileList


def main():

    #fracTimeOnTemp = []
    #with open('fracTimeOnkOn5.00kOff1.00
       numFil500filLen1numNets1numCargs100gamma0.80INS0.txt')
       as fp:
```

```
#    for line in fp:
#        fracTimeOnTemp.append([float(line)])



msdList1 = toList('MSDkOn5.00kOff1.00numFil1500filLen5.00
   numNets1numCargs10000gamma0.20INS0.txt')
msdList2 = toList('MSDkOn5.00kOff1.00numFil1500filLen5.00
   numNets1numCargs10000gamma0.40INS0.txt')
msdList3 = toList('MSDkOn5.00kOff1.00numFil1500filLen5.00
   numNets1numCargs10000gamma0.60INS0.txt')
msdList4 = toList('MSDkOn5.00kOff1.00numFil1500filLen5.00
   numNets1numCargs10000gamma0.80INS0.txt')
msdList5 = toList('MSDkOn5.00kOff1.00numFil1500filLen5.00
   numNets1numCargs10000gamma1.00INS0.txt')
msdListNone = toList('MSDkOn5.00kOff1.00
   numFil0filLen1numNets1numCargs10000gamma0.80INS0.txt')

msdSupers = [[0.9373,1.1104,1.2366,1.3364,1.3601],\
             [1.4047,1.6548,1.7047,1.7395,1.7574],\
             [1.6555,1.9016,1.8807,1.8396,1.7955],\
             [1.8357,1.9023,1.8193,1.8603,1.8777],\
             [1.8958,1.9081,1.9093,1.8619,1.8513]]
msdSubs = [[0.7518,0.7245,0.7081,0.7528,0.7228],\
           [0.7158,0.7067,0.7128,0.7972,0.7729],\
           [0.7391,0.7736,0.7769,0.8151,0.8426],\
           [0.7844,0.8100,0.8600,0.8637,0.9160],\
           [0.7886,0.8475,0.9036,0.8879,0.8956]]
fracTimeOn = [[0.0071,0.0097,0.0145,0.0177,0.0266],\
              [0.0099,0.0178,0.0347,0.0482,0.0719],\
              [0.0182,0.0397,0.0550,0.0593,0.1358],\
              [0.0316,0.0891,0.0787,0.1143,0.1584],\
              [0.0782,0.1739,0.2083,0.2666,0.3121]]

dtReg = (0.1*0.1) / (4*0.051)
time_values = np.arange(0,20000*dtReg,dtReg)
time_values2 = np.arange(0,2000*dtReg,dtReg)
msd_values1 = np.array(msdList1)
msd_values2 = np.array(msdList2)
msd_values3 = np.array(msdList3)
msd_values4 = np.array(msdList4)
msd_values5 = np.array(msdList5)
msd_valuesNone = np.array(msdListNone)
t = np.linspace(0,1000,1000-0)

divideTime = 3 / dtReg
```

```
dTime = int(divideTime)


fitParams,other = curve_fit(fitFunc,time_values[:dTime],
   msd_values4[:dTime])
fitParams2, other2 = curve_fit(fitFunc,time_values[dTime
   :],msd_values4[dTime:])
fitParamsNone, otherNone = curve_fit(fitFunc,time_values2,
   msd_valuesNone)
#print "short-time exponent", fitParams[1]
#print "long-time exponent", fitParams2[1]




#print 20/dtReg

#plt.ylim(.001,1000)
plt.xlim(.1,1000)
plt.xlabel('time (s)')
plt.ylabel('MSD ($\mu$m$^2$)')
#plt.plot(time_values,msd_values1,label='$\\alpha = 0.2$')
#plt.plot(time_values,msd_values2,label='$\\alpha = 0.4$')
#plt.plot(time_values,msd_values3,label='$\\alpha = 0.6$')
plt.scatter(time_values,msd_values4,label='$\\alpha = 0.8$
   ',color='red',marker='o')
#plt.plot(time_values,msd_values5,label='$\\alpha = 1.0$')
plt.scatter(time_values2,msd_valuesNone,marker='o')
#plt.legend(loc='upper left')
plt.plot(time_values,fitFunc(time_values,fitParams[0],
   fitParams[1]),color='blue',linestyle='dashed')
plt.plot(time_values,fitFunc(time_values,fitParams2[0],
   fitParams2[1]),color='green',linestyle='dashed')
plt.plot(time_values,fitFunc(time_values,fitParamsNone[0],
   fitParamsNone[1]),linestyle='dashed')
plt.xscale('log')
plt.yscale('log')

plt.show()


#x = dtReg * np.arange(time_values)

#print dtReg

#print len(msdList)
```

```
    #time_values = np.arange(len(msdList))




    #fracTimeOnList = []
    #for i in range(len(fracTimeOnTemp)):
    #    fracTimeOnList.append(fracTimeOnTemp[i][0])
    #
    #avgFracTimeOn = sum(fracTimeOnList) / len(fracTimeOnList)
    #print "average fraction of time on network: ",
       avgFracTimeOn



    #color maps
    #fig = plt.figure()
    #im = plt.imshow(fracTimeOn, origin = 'lower',extent =
       [100,500,1,5],aspect = 100)
    #im = plt.imshow(msdSupers, origin = 'lower', cmap = 'jet
       ',extent = [500,2500,1,5],aspect = 500,interpolation='
       bilinear')
    #im = plt.imshow(msdSubs, origin = 'lower',cmap = 'jet',
       extent = [500,2500,1,5],aspect = 500,interpolation='
       bilinear')
    #fig.colorbar(im)
    #plt.show()



main()
```

7.3.6 tamsdAnalysis.py

This program analyzes TA-MSD data.

```
# -*- coding: utf-8 -*-
"""
Created on Mon Apr 03 13:35:50 2017

@author: GPLP
"""

import matplotlib.pyplot as plt
```

```
import numpy as np
from scipy.optimize import curve_fit

def fitFunc(t, a):
    return a*t

def fitFuncPow(t,a,b):
    return a*t**(b)

def main():


    #with open('TimeMSDkOn5.00kOff1.00
       numFil300filLen5numNets1numCargs100gamma0.80INS0.txt')
       as file:
    #    ta_msdConstTArray = [[float(digit) for digit in line.
       split()] for line in file]

    with open('TAMSDkOn5.00kOff1.00numFil0filLen5.00
       numNets1numCargs1000gamma0.80INS0.txt') as file:
        ta_msdConstTArray1 = [[float(digit) for digit in line.
           split()] for line in file]

    with open('TAMSDs1kOn5.00kOff1.00numFil1500filLen5.00
       numNets10numCargs10gamma0.80INS0.txt') as file:
        ta_msdConstTArray2 = [[float(digit) for digit in line.
           split()] for line in file]
    with open('TAMSDs2kOn5.00kOff1.00numFil1500filLen5.00
       numNets10numCargs10gamma0.80INS0.txt') as file:
        ta_msdConstTArray3 = [[float(digit) for digit in line.
           split()] for line in file]
    with open('TAMSDs3kOn5.00kOff1.00numFil1500filLen5.00
       numNets10numCargs10gamma0.80INS0.txt') as file:
        ta_msdConstTArray4 = [[float(digit) for digit in line.
           split()] for line in file]
    #with open('WaitTimeMSDkOn5.00kOff1.00
       numFil1500filLen4numNets5numCargs100gamma0.80INS0.txt')
        as file:
    #    ta_msdConstTArray5 = [[float(digit) for digit in line
       .split()] for line in file]

    #fracTimeOnTemp = readlines(\
    #'fracTimeOnkOn5.00kOff1.00
       numFil300filLen5numNets1numCargs100gamma0.80INS0.txt')

    #fracTimeOnTemp = []
```

```
#with open('fracTimeOnkOn5.00kOff1.00
   numFil300filLen4numNets1numCargs100gamma0.80INS0.txt')
   as fp:
#    for line in fp:
#        fracTimeOnTemp.append([float(line)])

#max eb parameter (at 2000 seconds)
taConstT = [[0.0228,0.0319,0.0282,0.0505,0.0483],\
            [0.0354,0.0608,0.0511,0.1049,0.0751],\
            [0.0928,0.0981,0.1120,0.1295,0.0846],\
            [0.1519,0.0931,0.0967,0.1372,0.1337],\
            [0.1105,0.1428,0.1116,0.1523,0.1354]]

#decay exponent of constant wait time tamsd (s = 1 second)
taConstS = [[0.2687,0.2489,0.1849,0.2205,0.2363],\
            [0.2200,0.2193,0.2147,0.2513,0.1836],\
            [0.2308,0.0769,0.2030,0.1768,0.2100],\
            [0.2500,0.1993,0.1586,0.1786,0.0835],\
            [0.1921,0.1743,0.1551,0.2284,0.0648]]

fracTimeOn = [[0.0073,0.0117,0.0159,0.0339,0.0277],\
              [0.0108,0.0174,0.0536,0.0404,0.0502],\
              [0.0117,0.0571,0.0785,0.0835,0.1099],\
              [0.0241,0.0414,0.1101,0.1346,0.1905],\
              [0.1190,0.1023,0.2126,0.3053,0.3011]]

#plot of all trajectories
for i in range(len(ta_msdConstTArray1[0])):
    tempList = []
    for j in range(len(ta_msdConstTArray1)):
        tempList.append(ta_msdConstTArray1[j][i])
    #plt.plot(tempList)


#ta_msd2Array = np.zeros([len(ta_msdConstTArray1),len(
   ta_msdConstTArray1[0])])
#for i in range(len(ta_msdConstTArray1)):
#    print i
#    for j in range(len(ta_msdConstTArray1[0])):
#        ta_msd2Array[i,j] = ta_msdConstTArray1[i][j] ** 2

ta_msd2Array = []
for i in range(len(ta_msdConstTArray1)):
    ta_msd2Array.append([])
    for j in range(len(ta_msdConstTArray1[i])):
```

```
            ta_msd2Array[i].append(ta_msdConstTArray1[i][j
               ]**2)

    #print len(ta_msd2Array[1000])
    #print len(ta_msdConstTArray1[1000])



    #plot of average of all trajectories
    avgtempList1 = []
    avgtempList2 = []
    avgtempList3 = []
    avgtempList4 = []
    #avgtempList5 = []
    avg2tempList1 = []

    #standard deviation
    stdDevList = []

    for i in range(len(ta_msdConstTArray1)):

        avg1 = sum(ta_msdConstTArray1[i])/len(
           ta_msdConstTArray1[i])
        #avg2 = sum(ta_msdConstTArray2[i])/len(
           ta_msdConstTArray2[i])
        #avg3 = sum(ta_msdConstTArray3[i])/len(
           ta_msdConstTArray3[i])
        #avg4 = sum(ta_msdConstTArray4[i])/len(
           ta_msdConstTArray4[i])
        #avg5 = sum(ta_msdConstTArray5[i])/len(
           ta_msdConstTArray5[i])
        avgSq = sum(ta_msd2Array[i])/len(ta_msdConstTArray1[i
           ])

        avgtempList1.append(avg1)
        #avgtempList2.append(avg2)
        #avgtempList3.append(avg3)
        #avgtempList4.append(avg4)
        #avgtempList5.append(avg5)
        avg2tempList1.append(avgSq)
        tempSum = 0
        for j in range(len(ta_msdConstTArray1[i])):
            tempSum = tempSum + (ta_msdConstTArray1[i][j]-avg1
               )**2.0
        stdDev = (tempSum/len(ta_msdConstTArray1[i]))**(0.5)
        stdDevList.append(stdDev)
```

```
#plt.plot(avgtempList1)
#plt.plot(avgtempList2)
#plt.plot(avgtempList3)
#plt.plot(avgtempList4)
#plt.plot(avgtempList5)
#plt.plot(avg2tempList)
#plt.xscale('log')
#plt.yscale('log')


#ergodicity breaking
ebList = []
for i in range(1,len(avgtempList1)):
    eb = (avg2tempList1[i] - (avgtempList1[i])**2) / ((
       avgtempList1[i])**2)
    ebList.append(eb)

for i in range(1,len(ebList)):
    ebList[i] = ebList[i]/i

#print ebList[len(ebList)-1]

#no log
plt.xlabel('$\Delta$')
plt.ylabel('EB/$\Delta$')

#with log
#plt.xscale('log')
#plt.yscale('log')
plt.xlim(1,20)
#plt.xlabel('log($\Delta$)')
#plt.ylabel('log(EB/$\Delta}$)')

plt.plot(ebList,'o',label='Data')
#plt.plot(ebList,label='Guideline')


t = np.arange(1,len(ebList)+1,1)
plt.plot(t,ebList[1]/t,'--',label=str(round(ebList[1],2))
   +'$\Delta^{-1}$')
taFitParams, taFitCovs = curve_fit(fitFuncPow,t,ebList)
#print taFitParams
#plt.plot(t,taFitParams[0]*t**(taFitParams[1]),label='fit:
    '+str(round(taFitParams[0],2))+'$\Delta^{'+str(round(
   taFitParams[1],2))+'}$')
```

```
    plt.legend()




    #fit the data using scipy
    #t = np.linspace(1,len(avgtempList2),len(avgtempList2))
    #fitParams, fitCovariances = curve_fit(fitFunc,t[1:],
       avgtempList[1:])
    #fitParams, fitCovariances = curve_fit(fitFuncPow,t[100:],
       avgtempList2[100:])



    #plt.plot(fitFunc(t,fitParams[0]))
    #plt.plot(fitFuncPow(t,fitParams[0],fitParams[1]))
    #plt.errorbar(t,avgtempList,yerr=stdDevList)
    #print fitParams[1]
    #print fitCovariances
    #plt.xscale('log')
    #plt.yscale('log')


    #fracTimeOnList = []
    #for i in range(len(fracTimeOnTemp)):
    #    fracTimeOnList.append(fracTimeOnTemp[i][0])

    #avgFracTimeOn = sum(fracTimeOnList) / len(fracTimeOnList)
    #print "average fraction of time on network: ",
       avgFracTimeOn


    #color maps
    #fig = plt.figure()
    #im = plt.imshow(fracTimeOn, origin = 'lower',extent =
       [100,500,1,5],aspect = 100)
    #im = plt.imshow(taConstS, origin = 'lower', cmap = 'jet',
        extent = [100,500,1,5],aspect = 100)
    #fig.colorbar(im)

    plt.show()

main()
```

7.3.7 fptdFigures.py

This program analyzes FPTD data.

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 02 07:22:14 2017

@author: GPLP
"""
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
from scipy.stats import chisquare

def fitFuncExp(t,a,b):
    return a*np.exp(-b*t)

def fitFuncPow(t,a,b):
    return a*t**(b)

def fitFuncPowCDF(t,a,b,c):
    return (a + b*t**(c))

def readFile(aFile):
    with open(aFile) as file:
        fileList = [[float(digit) for digit in line.split()]
           for line in file]
    return fileList

def main():


    #fracTimeOnTemp = readFile(\
    #'fracTimeOnkOn5.00kOff1.00
       numFil300filLen5numNets1numCargs10000gamma0.80INS1.txt
       ')

    fptdExpos = [[0.2324,0.3039,0.3103,0.3697,0.3948],\
                 [0.4374,0.5563,0.6428,0.6999,0.7255],\
                 [0.6408,0.7697,0.8684,0.9224,0.9193],\
                 [0.7934,0.9420,0.9788,1.0749,1.0507],\
                 [0.9216,1.0728,1.0947,1.1771,1.1668]]
    #fracTimeOn = [[0.0045,0.0106,0.0150,0.0266,0.0439],\
    #              [0.0187,0.0472,0.1107,0.1088,0.1566],\
    #              [0.0736,0.1563,0.1902,0.2811,0.2260],\
    #              [0.0896,0.2162,0.2559,0.3526,0.3753],\
```

```
#              [0.1797,0.3239,0.3861,0.4755,0.4724]]

#fracTimeOnList = []
#for i in range(len(fracTimeOnTemp)):
#    fracTimeOnList.append(fracTimeOnTemp[i][0])

#avgFracTimeOn = sum(fracTimeOnList) / len(fracTimeOnList)
#print "average fraction of time on network: ",
   avgFracTimeOn

tempList1 = []
tempList2 = []
tempList3 = []
tempList4 = []
tempList5 = []



l1 = readFile('FPTDkOn5.00kOff1.00numFil300filLen5.00
   numNets1numCargs10000gamma0.20INS1.txt')
l2 = readFile('FPTDkOn5.00kOff1.00numFil300filLen5.00
   numNets1numCargs10000gamma0.40INS1.txt')
l3 = readFile('FPTDkOn5.00kOff1.00numFil300filLen5.00
   numNets1numCargs10000gamma0.60INS1.txt')
l4 = readFile('FPTDkOn5.00kOff1.00numFil300filLen5.00
   numNets100numCargs100gamma0.80INS1.txt')
l5 = readFile('FPTDkOn5.00kOff1.00numFil300filLen5.00
   numNets1numCargs10000gamma1.00INS1.txt')

for i in range(len(l1)):
    tempList1.append(l1[i][0])
    tempList2.append(l2[i][0])
    tempList3.append(l3[i][0])
    tempList4.append(l4[i][0])
    tempList5.append(l5[i][0])


lowerBound = 10
upperBound = 1000

#curve fitting
t = np.arange(lowerBound,upperBound)
fitParams1,fitCovariances1 = curve_fit(fitFuncPow,t,
   tempList1[lowerBound:upperBound])
```

```
fitParams2,fitCovariances2 = curve_fit(fitFuncPow,t,
   tempList2[lowerBound:upperBound])
fitParams3,fitCovariances3 = curve_fit(fitFuncPow,t,
   tempList3[lowerBound:upperBound])
fitParams4,fitCovariances4 = curve_fit(fitFuncPow,t,
   tempList4[lowerBound:upperBound])



#quality of the fits
#print chisquare(tempList[10:1000],[fitFuncExp(i,fitParams
   [0],fitParams[1]) for i in range(10,1000)])
#print chisquare(tempList[10:1000],[fitFuncPow(i,
   fitParams2[0],fitParams2[1]) for i in range(10,1000)])

#print "'decaying' exponent",fitParams1[1]
#print "'decaying' exponent",fitParams2[1]
#print "'decaying' exponent",fitParams3[1]
print "'decaying' exponent",fitParams4[1]


fitParams5,fitCovariances5 = curve_fit(fitFuncExp,t,
   tempList5[lowerBound:upperBound])

# calculating cdf for alpha = 0.8 fptd
sumCdf = 0
cdf = []
for i in range(len(tempList4)):
    tempList4[i] /= 10000
    sumCdf += tempList4[i]
    cdf.append(sumCdf)

plt.plot(t,cdf[lowerBound:upperBound])


# fitting the cdf
fitParamsC,fitCovariancesC = curve_fit(fitFuncPowCDF,t,cdf
   [lowerBound:upperBound],maxfev=100000)

# linear scatter plots
#plt.scatter(t,tempList1[lowerBound:upperBound],marker='o
   ')
#plt.scatter(t,tempList2[lowerBound:upperBound],marker='o
   ')
#plt.scatter(t,tempList3[lowerBound:upperBound],marker='o
   ')
```

```
#plt.scatter(t,tempList4[lowerBound:upperBound],marker='o
    ')
#plt.scatter(t,tempList5[lowerBound:upperBound],marker='o
    ')

print "'decaying' exponent",fitParamsC[0],fitParamsC[1],
    fitParamsC[2]


plt.plot(t,fitFuncPowCDF(t,fitParamsC[0],fitParamsC[1],
    fitParamsC[2]),label='$\\alpha = 0.8$')




#plt.plot(t,fitFuncPow(t,fitParams1[0],fitParams1[1]),
    label='$\\alpha = 0.2$')
#plt.plot(t,fitFuncPow(t,fitParams2[0],fitParams2[1]),
    label='$\\alpha = 0.4$')
#plt.plot(t,fitFuncPow(t,fitParams3[0],fitParams3[1]),
    label='$\\alpha = 0.6$')
#plt.plot(t,fitFuncPow(t,fitParams4[0],fitParams4[1]),
    label='$\\alpha = 0.8$')
#plt.plot(t,fitFuncExp(t,fitParams5[0],fitParams5[1]),
    label='$\\alpha = 1.0$')
plt.legend(loc='upper right')

plt.ylabel('FPTD (counts)')
plt.xlabel('time (s)')
#plt.xlim(10,1000)
#plt.ylim(1,400)
#plt.legend()
#plt.xscale('log')
#plt.yscale('log')
plt.show()



#colormaps
#fig = plt.figure()
#im = plt.imshow(fracTimeOn, origin = 'lower',extent =
    [100,500,1,5],aspect = 100)
#im = plt.imshow(fptdExpos, origin = 'lower',extent =
    [100,500,1,5],aspect = 100)
```

```
    #fig.colorbar(im)


main()
```

7.3.8 colorMapGeneral3.py

This program constructs the color maps of the data analyzed previously. These are the color maps that are found in the chapter on anomalous transport.

```
import matplotlib.pyplot as plt

#interpolation function
def bilinearInterpolation(x,y,vals2D):
    xmax = 500.0
    ymax = 5.0
    xfact = 100.0
    yfact = 1.0
    x1 = x // 100 * 100.0
    x2 = x1 + xfact
    y1 = y // 1 * 1.0
    y2 = y1 + yfact
    i = int(y/yfact) - 1
    j = int(x/xfact) - 1
    fxy1 = (x2-x)/(x2-x1)*vals2D[i][j]+(x-x1)/(x2-x1)*vals2D[i
       ][j+1]
    fxy2 = (x2-x)/(x2-x1)*vals2D[i+1][j]+(x-x1)/(x2-x1)*vals2D
       [i+1][j+1]
    fxy = (y2-y)/(y2-y1)*fxy1 + (y-y1)/(y2-y1)*fxy2
    return fxy


#interpolation function
#mass input
def bilinInterpMass(mass,vals2D):
    xfact = 100.0
    yfact = 1.0
    lengths = []
    z = []
    if (mass >= 500):
        minLen = min(mass/499.0, 4.9)
        maxLen = 4.9
    else:
        minLen = mass/(mass-1)
        maxLen = mass/101
    x = mass / minLen
```

```
    y = minLen
    dy = (maxLen - minLen) / 40
    while(y <= maxLen):
        x1 = x // 100 * 100.0
        x2 = x1 + xfact
        y1 = y // 1 * 1.0
        y2 = y1 + yfact
        i = int(y/yfact) - 1
        j = int(x/xfact) - 1
        fxy1 = (x2-x)/(x2-x1)*vals2D[i][j]+(x-x1)/(x2-x1)*
           vals2D[i][j+1]
        fxy2 = (x2-x)/(x2-x1)*vals2D[i+1][j]+(x-x1)/(x2-x1)*
           vals2D[i+1][j+1]
        fxy = (y2-y)/(y2-y1)*fxy1 + (y-y1)/(y2-y1)*fxy2
        z.append(fxy)
        lengths.append(y)
        y = y + dy
        x = mass / y
    return lengths, z
```

```
def main():

    #average flux out by 10s
    out10avg = [[0.570000, 0.750000, 1.030000, 1.240000,
       1.610000],
                [2.220000, 3.550000, 4.880000, 6.330000,
                   7.290000],
                [4.860000, 8.260000, 11.010000, 12.040000,
                   14.350000],
                [8.970000, 13.710000, 17.190000, 17.470000,
                   19.180000],
                [14.110000, 19.700000, 22.640000, 24.470000,
                   25.150000]]
    #average flux out by 100s
    out100avg = [[3.810000, 4.880000, 6.010000, 7.270000,
       8.930000],
                 [10.340000, 16.120000, 22.660000, 28.290000,
                    34.490000],
                 [20.480000, 32.050000, 42.490000, 47.760000,
                    54.260000],
```

```
            [33.460000, 46.300000, 53.610000, 56.590000,
               62.780000],
            [49.250000, 62.170000, 69.600000, 74.240000,
               76.710000]]
#average flux out after 100s
out100avg2 = [[],[],[],[],[]]
for i in range(len(out100avg)):
    for j in range(len(out100avg[i])):
        out100avg2[i].append(100-out100avg[i][j])


#MFPT data

#overall MFPT
mfpts = [[20901.496980, 20912.001505, 19604.040386,
   17845.361140, 20277.588450],
         [15536.934338, 12104.430417, 8191.819561,
            6472.469627, 5488.723420],
         [10198.689438, 6028.738413, 4113.791565,
            2589.134276, 2472.403544],
         [5175.396216, 2697.863910, 1967.081567,
            1553.866925, 1375.408970],
         [2689.650648, 1147.063759, 680.370400,
            533.215085, 473.183413]]

#MFPT standard deviation
mfptStdevs = [[5755.898657, 9425.448567, 9927.163744,
   9350.076329, 21568.708661],
            [10288.034119, 10280.942310, 7523.041154,
               5793.490683, 5366.468494],
            [9428.787124, 4734.632566, 5864.933007,
               2974.841794, 4428.577627],
            [4848.489275, 2286.449963, 2466.075192,
               1383.737160, 1275.501484],
            [2568.826398, 1491.681241, 724.067629,
               732.567479, 714.767678]]

#average of standard deviations
stdevAvg = [[82905.511837, 84366.079882, 79465.100796,
   74826.966022, 80603.735198],
           [69016.131075, 59398.646737, 45309.271922,
              42429.944821, 36054.621695],
           [52482.269621, 38666.404843, 29373.372678,
              19616.937224, 20491.743505],
           [33619.798949, 21004.022667, 15637.889841,
              14436.379428, 14009.293537],
```

```
            [22740.830746, 9632.788240, 7654.654856,
               6052.643270, 5536.412024]]


#normalizing standard deviations
for i in range(len(mfpts)):
    for j in range(len(mfpts[i])):
        mfptStdevs[i][j] = mfptStdevs[i][j] / mfpts[i][j]
        #stdevAvg[i][j] = stdevAvg[i][j] / mfpts[i][j]



#variation in flux out by 10s
out10stdev = [[0.710704, 0.804674, 1.108648, 1.068831,
   1.232031],
              [1.285146, 1.981792, 2.141401, 2.724170,
                 2.627908],
              [2.015043, 3.035194, 3.477053, 3.557865,
                 3.683409],
              [3.247938, 3.945364, 3.786014, 4.863034,
                 4.222274],
              [3.652109, 4.659399, 5.231673, 4.484317,
                 5.286540]]
#variation in flux out by 100s
out100stdev = [[1.863840, 2.173845, 2.643842, 3.062205,
   3.332432],
               [3.782116, 4.964434, 5.625336, 5.998825,
                  7.047688],
               [6.004132, 6.973342, 7.381727, 6.858746,
                  7.025126],
               [7.003456, 8.386298, 7.123054, 7.802685,
                  7.017948],
               [8.157665, 8.142549, 8.238932, 7.190438,
                  6.202088]]


#msd at 10s
msd10 = [[69.581944/1976, 88.794141/2109, 111.229480/2137,
    123.316868/2195, 135.102968/2325],
         [73.430564/1932, 93.515927/2033, 109.623798/2104,
             125.049170/2138, 130.210496/2290],
         [72.546248/1936, 87.823341/2024, 110.559687/2185,
             114.582143/2144, 137.155808/2300],
         [70.694660/1908, 92.282295/2019, 106.792043/2145,
             121.891740/2217, 126.036580/2221],
```

```
          [73.714240/2041, 86.075748/1981, 104.372972/2137,
              122.103398/2182, 140.465580/2275]]


#msd at 100s
msd100 = [[294.258703/1101, 540.157455/1193,
   672.831761/1252, 826.618722/1326, 1052.494706/1437],
           [722.093250/1126, 1298.696021/1310,
              1989.772316/1432, 2461.064212/1550,
              2660.435852/1603],
           [1258.417762/1165, 2127.599265/1335,
              3141.687283/1521, 3606.934766/1567,
              4920.205895/1829],
           [1544.690158/1182, 2690.336980/1361,
              3502.937448/1515, 4397.074885/1616,
              4865.776005/1682],
           [1661.178566/1233, 3128.072986/1422,
              3928.799090/1506, 4576.097542/1620,
              5594.247695/1737]]


#variaton in msd at 10s
msd10stdev = [[0.177165, 0.277559, 0.316284, 0.358758,
   0.376862],
              [0.420243, 0.726115, 0.676183, 0.903460,
                 0.926523],
              [0.775392, 1.238581, 1.319257, 1.319592,
                 1.517769],
              [1.146256, 1.619773, 1.738303, 2.031710,
                 2.023696],
              [1.506506, 1.961689, 2.099387, 2.018767,
                 2.043376]]
#variation in msd at 100s
msd100stdev = [[0.898412, 1.375464, 1.773565, 1.914777,
   2.521721],
               [2.732093, 4.268729, 5.154552, 5.722110,
                  6.081641],
               [4.935392, 7.925013, 10.237778, 10.749308,
                  11.285958],
               [7.107889, 11.541396, 12.936895, 16.750645,
                   15.987546],
               [12.073025, 14.428583, 16.574697,
                  17.149316, 16.381906]]

#normalizing standard deviations
```

```
for i in range(len(msd10)):
    for j in range(len(msd10[i])):
        msd10stdev[i][j] = msd10stdev[i][j] / msd10[i][j]
        msd100stdev[i][j] = msd100stdev[i][j] / msd100[i][
           j]


#fptd exponents
fptdExpos = [[0.2324,0.3039,0.3103,0.3697,0.3948],\
             [0.4374,0.5563,0.6428,0.6999,0.7255],\
             [0.6408,0.7697,0.8684,0.9224,0.9193],\
             [0.7934,0.9420,0.9788,1.0749,1.0507],\
             [0.9216,1.0728,1.0947,1.1771,1.1668]]
#msd exponents
msdSupers = [[0.9373,1.1104,1.2366,1.3364,1.3601],\
             [1.4047,1.6548,1.7047,1.7395,1.7574],\
             [1.6555,1.9016,1.8807,1.8396,1.7955],\
             [1.8357,1.9023,1.8193,1.8603,1.8777],\
             [1.8958,1.9081,1.9093,1.8619,1.8513]]
msdSubs = [[0.7518,0.7245,0.7081,0.7528,0.7228],\
           [0.7158,0.7067,0.7128,0.7972,0.7729],\
           [0.7391,0.7736,0.7769,0.8151,0.8426],\
           [0.7844,0.8100,0.8600,0.8637,0.9160],\
           [0.7886,0.8475,0.9036,0.8879,0.8956]]

#colormaps
fig = plt.figure()
#fptd colormaps
im = plt.imshow(stdevAvg, origin = 'lower',cmap = 'jet',
   extent = [100,500,1,5],aspect = 100,interpolation='
   bilinear')
#msd colormaps
#im = plt.imshow(msd100stdev, origin = 'lower',cmap = 'jet
   ',extent = [500,2500,1,5],aspect = 500,interpolation='
   bilinear')
clb = fig.colorbar(im)
#clb.ax.set_title('$\sigma_{MSD}$(100s) / MSD(100s)')
plt.xlabel('number of filaments')
plt.ylabel('filament length ($\mu$m)')


l1500, vals1500 = bilinInterpMass(1500.0,fptdExpos)
l1000, vals1000 = bilinInterpMass(1000.0,fptdExpos)
l800, vals800 = bilinInterpMass(800.0,fptdExpos)
l600, vals600 = bilinInterpMass(600.0,fptdExpos)
l500, vals500 = bilinInterpMass(500.0,fptdExpos)
```

```
    l400, vals400 = bilinInterpMass(400.0,fptdExpos)
    l300, vals300 = bilinInterpMass(300.0,fptdExpos)

    #interpolation attempt
    #plotting constant mass interpolations
    #plt.plot(l1500,vals1500,label='Mass = 1500',marker='o')
    #plt.plot(l1000,vals1000,label='Mass = 1000',marker='o')
    #plt.plot(l800,vals800,label='Mass = 800',marker='o')
    #plt.plot(l600,vals600,label='Mass = 600',marker='o')
    #plt.plot(l500,vals500,label='Mass = 500',marker='o')
    #plt.plot(l400,vals400,label='Mass = 400',marker='o')
    #plt.plot(l300,vals300,label='Mass = 300',marker='o')


    #plt.legend(loc = 'lower right')
    #plt.xlabel('filament length ($\mu$m)')
    #plt.ylabel('FPTD exponents')

    plt.show()


main()
```

7.4 Cargo Simulations On Real Networks Programs

7.4.1 realFils.py

Converts the FIRE algorithm output to a more useable txt file.

```
# this program makes the filaments more easily readable by a C
    or C++ program
def main():

    # open the file containing the filament vertex data
    with open('filaments.txt') as file:
        filaments = [[int(number) for number in line.split()]
           for line in file]

    # only select lines having a clear list of vertices for
       each filament
    filaments = [x for x in filaments if x]

    # write the now more organized filament vertex data to a
       new file
    with open('filamentsBetter.txt','w') as file:
        for fil in filaments:
```

```
            file.write(" ".join(str(v) for v in fil) + '\n')


main()
```

7.4.2 simRealNetOnOnly.cpp

This program simulates the movement of cargos over a network obtained through implementing the FIRE algorithm on a network image.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

// to use c++ vectors
#include <vector>
#include <algorithm>
using namespace std;

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

// global variables

// number of vertices in the network
int numVertices;

// for storing (x, y) vertex positions
vector<double> xVertexPositions;
vector<double> yVertexPositions;

// maximum x and y values for the
// transport region
double xMax, yMax;

// for storing filaments and their vertices
vector<vector<int> > filaments;

// for storing vertices and their filaments
vector<vector<int> > vertices;

// number of filaments in the network
```

```
int numFilaments;

// number of cargos used in the simulation
int numCargos = 1000;

// cargo radius in micrometers (um)
double cRad = 0.1;

// cargo speed while on a filament (um / s)
double v = 1.0;

// cargo step size (um)
double dstep = 0.1;

// cargo (varying) time step
double dt;

// cargo timestep off the network (diffusion phase)
double dtOff;

// cargo timestep on the network (ballistic motion phase)
double dtOn;

// amount of time that has passed (s)
double t;

// maximum measuring time (s)
double tMax = 100.0;

// diffusion constant (um / s^2)
double D = 0.051;

// cargo attachment and detachment rates (s^-1)
double kOn = 10000.0, kOff = 0.02;

// cargo switching rate
//double kSwitch = 5.0;

// cargo switching rate in terms of probability
double switchProb = 0.0;

// is the cargo off or on a filament?
int OFF, ON;

// used to output to different files
char sBeg[500];
```

```
char sEnd[500];

int main(){

 // read and store the vertex positions
 FILE *inpVerts = fopen("vertexPositions1to10.txt", "r");

 // (x, y) coordinates of current vertex
 double xVP, yVP;

 // micrometers per pixel in the network images
 double umPerPixel = 0.0675;

 // vertex "0" is at position (0.0, 0.0)
 xVertexPositions.push_back(0.0);
 yVertexPositions.push_back(0.0);

 // scan x and y positions of each vertex until the end of the
     file is reached
 // convert position indices (image pixels) to micrometers
 while (fscanf(inpVerts, "%lf %lf", &xVP, &yVP) != EOF){
  xVP *= umPerPixel;
  yVP *= umPerPixel;
  xVertexPositions.push_back(xVP);
  yVertexPositions.push_back(yVP);
  //fprintf(outp1,"%lf\t%lf\n", xVP, yVP);
 }

 // close the file from which vertex positions were received
 fclose(inpVerts);

 // get the number of vertices
 numVertices = xVertexPositions.size();

 // calculate the max x and y values
 xMax = *max_element(xVertexPositions.begin(),
    xVertexPositions.end());
 yMax = *max_element(yVertexPositions.begin(),
    yVertexPositions.end());

 // each vertex belongs to a certain number of filaments
 // vertices index "i" corresponds to vertex number "i"
 vector<int> currentVertex;

 // prepare the filament vector for each vertex
 for (int i = 0; i < numVertices; i++){
```

```
 vertices.push_back(currentVertex);
}

// read and store filaments and their vertices
FILE *inpFils = fopen("filamentsBetter1to10.txt", "r");

// variables used in storing filament vertices
int valueInt;
char spaceChar;
vector<int> currentFilament;

// filament "0" contains only vertex "0"
currentFilament.push_back(0);
filaments.push_back(currentFilament);
currentFilament.clear();

while(fscanf(inpFils, "%d%c", &valueInt, &spaceChar) != EOF){
 // add the scanned number to the current filament vertex
    vector
 currentFilament.push_back(valueInt);
 // new line
 // end of the current filament has been reached
 if(spaceChar == '\n'){
  // add the current filament to the vector of filaments
  filaments.push_back(currentFilament);
  // prepare to start reading the vertex numbers for the next
      filament
  currentFilament.clear();
 }

}

// close the file from which filament vertices were received
fclose(inpFils);

// calculate the number of filaments in the network
numFilaments = filaments.size();

// to calculate the number of vertices in the current
   filament
int numFilVerts;

// add filaments to each vertex vector
// and calculate filament lengths

FILE *outpLengths = fopen("filamentLengths.txt","w");
```

```
double currentFilLength;
for (int i = 0; i < numFilaments; i++){
 currentFilLength = 0.0;
 numFilVerts = filaments[i].size();
 for (int j = 0; j < numFilVerts; j++){
  vertices[filaments[i][j]].push_back(i);
 }
 for (int j = 1; j < numFilVerts; j++){
  currentFilLength += sqrt(pow((xVertexPositions[filaments[i
     ][j]]-xVertexPositions[filaments[i][j-1]]),2) +
         pow((yVertexPositions[filaments[i][j]]-
            yVertexPositions[filaments[i][j-1]]),2));
 }
 fprintf(outpLengths,"%lf\n",currentFilLength);
}

fclose(outpLengths);

// network should be set up now

// testing
//for (int i = 0; i < numVertices; i++){
// fprintf(outp1,"%lf %lf\n", xVertexPositions[i] /
   umPerPixel, yVertexPositions[i] / umPerPixel);
//}

//for (int i = 0; i < numFilaments; i++){
// numFilVerts = filaments[i].size();
// for (int j = 0; j < numFilVerts; j++){
//  fprintf(outp1,"%d ", filaments[i][j]);
// }
// fprintf(outp1,"\n");
//}

// to calculate the number of filaments connected to the
   current vertex
int numVertFils;
//for (int i = 0; i < numVertices; i++){
// numVertFils = vertices[i].size();
// for (int j = 0; j < numVertFils; j++){
//  fprintf(outp1,"%d ", vertices[i][j]);
// }
// fprintf(outp1,"\n");
//}
```

```
// time step while off the network



// calculate average distance between filament intersections
double avgIntSep = 0.0;
double intDistSum = 0.0;
int numIntCalcs = 0;


//create a vector of intersection vertices
vector<int> intersections;
for(int i = 0; i < numVertices; i++){
 numVertFils = vertices[i].size();
 if(numVertFils >= 2){
  intersections.push_back(i);
  // checking vertices
  //printf("vertex %d: ", i);
  //for(int j = 0; j < numVertFils; j++){
  // printf("%d ", vertices[i][j]);
  //}
  //printf("\n");

 }
}

// check the vertices that are intersections
//for(int i = 0; i < intersections.size(); i++){
// printf("%d\n", intersections[i]);
//}

FILE *outpDistances = fopen("intersectionDistances.txt","w");

// go through each filament, vertex by vertex
for(int i = 0; i < numFilaments; i++){
 numFilVerts = filaments[i].size();
 int firstInt = 0, secondInt = 0;
 for(int j = 0; j < numFilVerts; j++){
  // if a vertex is an intersection, look for the next one in
      the same filament
  if(firstInt && !secondInt){
   if(find(intersections.begin(), intersections.end(),
      filaments[i][j]) != intersections.end()){
    secondInt = filaments[i][j];
   }
```

```
  }
  // calculate distance between intersections and keep track
     of the number of calculations
  if(firstInt && secondInt){
   // check the two current intersections (on the same
      filament)
   //printf("firstInt: %d\tsecondInt: %d\n", firstInt,
      secondInt);
   double intDist = sqrt(pow((xVertexPositions[secondInt]-
      xVertexPositions[firstInt]),2) + pow((yVertexPositions[
      secondInt]-yVertexPositions[firstInt]),2));
   fprintf(outpDistances,"%lf\n",intDist);
   intDistSum += intDist;
   numIntCalcs++;
   // reset, look for two more intersections on the same
      filament
   firstInt = 0;
   secondInt = 0;
  }
  if(!firstInt){
   if(find(intersections.begin(), intersections.end(),
      filaments[i][j]) != intersections.end()){
    firstInt = filaments[i][j];
   }
  }

 }
}

fclose(outpDistances);


// calculate the average intersection separation distance
avgIntSep = intDistSum / numIntCalcs;
//printf("%.3lf\n", avgIntSep);


// begin placing cargos randomly throughout the network and
   let them start moving
srand((unsigned)time(NULL));

// set time step sizes
dtOff = pow(dstep, 2) / (4 * D);
dtOn = dstep / v;

// cargo position
```

```
double xc, yc;

// cargo direction of motion
double phi;

while(switchProb <= 1.0){


// for storing cargo data in a file
FILE *outp1;

sprintf(sBeg, "multipleCargoTrajectories");
sprintf(sEnd, "NumCargos%dSwitchProb%.3lfkOff%.3lf.txt",
   numCargos, switchProb, kOff);
outp1 = fopen(strcat(sBeg,sEnd), "w");

fprintf(outp1, "x:\t\ty:\t\tt:\t\tdt:\t\tON?\tFilament:\
   tMoving backwards?\tToward vertex:\t\tCargo Number:\n");

// start randomly placing cargos and letting them move
for(int i = 0; i < numCargos; i++){
 //fprintf(outp1,"A new cargo will start moving\n");

 // cargo begins in a random position at t = 0.0
 xc = xMax * (double)rand()/(RAND_MAX);
 yc = yMax * (double)rand()/(RAND_MAX);
 t = 0.0;
 // each cargo begins in the "off" state
 OFF = 1;
 ON = 0;
 dt = dtOff;


 // for keeping track of what filament the cargo ends up
    binding to
 int currentFil;
 int currentV1, currentV2;
 int vi;

 // used in calculated the distance from the cargo to nearby
    filaments
 double d;
 double minx1x2, maxx1x2, miny1y2, maxy1y2;
```

```
// by default cargo will move forward along a filament
int BACKWARD = 0;

while(xc < xMax && xc > 0.0 && yc < yMax && yc > 0.0 && t <
   tMax){



 // current filament segment vertices' positions
 double x1, y1, x2, y2;

 if(OFF and !ON){
 // start looking for filaments
 int f = 1;



 while (f < numFilaments && !ON && OFF){

  // look for filament segments
  int vert = 0;

  // current filament segment vertices
  int vert1, vert2;

  while (vert < filaments[f].size() - 1 && !ON && OFF){
   // the vertices that make up the current segment
   // (segment endpoints)
   vert1 = filaments[f][vert];
   vert2 = filaments[f][vert+1];

   // the positions of the vertices of the current segment
   // (endpoint positions)
   x1 = xVertexPositions[vert1];
   y1 = yVertexPositions[vert1];
   x2 = xVertexPositions[vert2];
   y2 = yVertexPositions[vert2];

   // distance from the cargo to the filament segment
   d = fabs((y2-y1)*xc-(x2-x1)*yc+x2*y1-y2*x1)/
    sqrt(pow((y2-y1),2)+pow((x2-x1),2));

   // min and max coordinates
   // (sets the filament segment "window" in 2D space)
   minx1x2 = 0.5*fabs(x1+x2)-0.5*fabs(x1-x2);
   maxx1x2 = 0.5*fabs(x1+x2)+0.5*fabs(x1-x2);
```

```
miny1y2 = 0.5*fabs(y1+y2)-0.5*fabs(y1-y2);
maxy1y2 = 0.5*fabs(y1+y2)+0.5*fabs(y1-y2);

// if the cargo is close enough to the filament segment,
// it has a chance of attaching
if(xc > minx1x2 && xc < maxx1x2 && yc > miny1y2 && yc <
   maxy1y2 && d < cRad){

  //probability of attaching to the network
  if((double)rand()/RAND_MAX <= (kOn*dt)){

   ON = 1;
   currentFil = f;


   // determine if cargo will move backward along the
      filament
   BACKWARD = rand() % 2;

   if(BACKWARD){
    currentV1 = vert2;
    currentV2 = vert1;
    vi = vert;
    //fprintf(outp1,"Cargo will now move backwards\n");
   } else {
    currentV1 = vert1;
    currentV2 = vert2;
    vi = vert + 1;
    //fprintf(outp1,"Cargo will now move forwards\n");
   }



   //fprintf(outp1,"Attached to filament %d, moving
      towards vertex %d\n", f, currentV2);
   //fprintf(outp1,"Distance to vertex %d: %lf\n",
      currentV2, sqrt(pow((yVertexPositions[currentV2]-yc
      ), 2) + pow((xVertexPositions[currentV2]-xc), 2)));

   //fprintf(outp1,"filament number %d: %d --> %d ", f,
      vert1, vert2);
  }


}
//fprintf(outp1,"%d --> %d ", vert1, vert2);
```

```
  // start checking proximity of next segment
  vert++;
 }
 //fprintf(outp1,"\n");
 // start checking proximity of next filament
 f++;

}
}

if(ON && !OFF){

 // if cargo is close enough to a vertex, allow switching
    if possible
 if(cRad > sqrt(pow((yVertexPositions[currentV2]-yc), 2) +
    pow((xVertexPositions[currentV2]-xc), 2))){
  //fprintf(outp1,"Near vertex %d\n", currentV2);
  // look for other filaments (if at a filament
     intersection)
  int fSwitch = 0;
  int SWITCHED = 0;
  while(fSwitch < vertices[currentV2].size() && !SWITCHED){
   // check for any filaments except the one the cargo is
      currently on
   if(vertices[currentV2][fSwitch] != currentFil){
    //fprintf(outp1,"Also a part of filament %d\n",
       vertices[currentV2][fSwitch]);
    // cargo might switch to another filament
    //fprintf(outp1,"Has a probability of switching to it
       of %lf\n", switchProb);
    if((double)rand()/RAND_MAX <= (switchProb)){
     // cargo has switched to another filament
     SWITCHED = 1;
     currentFil = vertices[currentV2][fSwitch];
     //fprintf(outp1,"Cargo has switched to filament %d
        which has vertices:\n", currentFil);

     //for(int newVert = 0; newVert < filaments[currentFil
        ].size(); newVert++){
     // fprintf(outp1,"%d\n", filaments[currentFil][newVert
        ]);
     //}

     // cargo might move backward on the new filament
     BACKWARD = rand() % 2;
```

```
// need to get new vertex to move towards
for(int newVert = 0; newVert < filaments[currentFil].
   size(); newVert++){

 // found the current vertex
 if(currentV2 == filaments[currentFil][newVert]){
  // cargo will move backward toward the previous
     vertex
  if(BACKWARD){
   // if this is the first vertex in the filament
   // fall off the filament
   if(currentV2 == filaments[currentFil][0]){
    OFF = 1;
    ON = 0;
    dt = dtOff;
    BACKWARD = 0;
    //fprintf(outp1,"Walked off filament %d at vertex
       %d\n", currentFil, currentV2);
    break;
   } else {
    currentV2 = filaments[currentFil][newVert-1];
    vi = newVert - 1;
    break;
   }

  // cargo will move forwards toward the next vertex
  } else {
   // if this is the last vertex in the filament
   // fall off the filament
   if(currentV2 == filaments[currentFil][filaments[
      currentFil].size()-1]){
    OFF = 1;
    ON = 0;
    dt = dtOff;
    BACKWARD = 0;
    //fprintf(outp1,"Walked off filament %d at vertex
       %d\n", currentFil, currentV2);
    break;
   } else {
    currentV2 = filaments[currentFil][newVert+1];
    vi = newVert + 1;
    break;
   }

  }
 }
```

```
    }

    //if(BACKWARD){
    // fprintf(outp1,"Will move backwards towards vertex %
       d\n", currentV2);
    //} else {
    // fprintf(outp1,"Will move forwards towards vertex %d
       \n", currentV2);
    //}

   }
  }
  fSwitch++;
 }
}

// cargo can walk off the filament
// if cargo is moving forward:
if((currentV2 == filaments[currentFil][filaments[
   currentFil].size()-1]) && !BACKWARD){
 if(cRad > sqrt(pow((yVertexPositions[currentV2]-yc), 2) +
     pow((xVertexPositions[currentV2]-xc), 2))){
  OFF = 1;
  ON = 0;
  dt = dtOff;
  BACKWARD = 0;
  // cargo fell off -- end the simulation
  break;
  //fprintf(outp1,"Walked off filament %d at vertex %d\n",
      currentFil, currentV2);
 }
}
// if cargo is moving backward:
if((currentV2 == filaments[currentFil][0]) && BACKWARD){
 if(cRad > sqrt(pow((yVertexPositions[currentV2]-yc), 2) +
     pow((xVertexPositions[currentV2]-xc), 2))){
  OFF = 1;
  ON = 0;
  dt = dtOff;
  BACKWARD = 0;
  // cargo fell off -- end the simulation
  break;
  //fprintf(outp1,"Walked off filament %d at vertex %d\n",
      currentFil, currentV2);
 }
}
```

```
 // allow possibility of detaching from the filament
 if((double)rand()/RAND_MAX <= (kOff * dt)){
  OFF = 1;
  ON = 0;
  dt = dtOff;
  BACKWARD = 0;
  break;
  //fprintf(outp1,"Detached from filament %d\n", currentFil
     );
 }
}

// make sure cargo is indeed on a filament
if(ON && OFF){
 OFF = 0;
 dt = dtOn;
}


// if off a filament, pick a random direction to travel in
if (OFF && !ON) {
 currentV2 = 0;
 currentFil = 0;
 phi = (2*M_PI)*(double)rand()/RAND_MAX;
 xc += dstep * cos(phi);
 yc += dstep * sin(phi);
 t += dt;
 //fprintf(outp1,"x: %lf\ty: %lf\tdt:%lf (Off)\n", xc, yc,
    dt);
 //fprintf(outp1,"Off the network\n");
}// end "off" movement

if (ON && !OFF){
 // cargo has approached the next vertex
 if(cRad > sqrt(pow((yVertexPositions[currentV2]-yc), 2) +
    pow((xVertexPositions[currentV2]-xc), 2))){
   // if at the last vertex in the filament
   if(BACKWARD){
    if(currentV2 == filaments[currentFil][0]){
     OFF = 1;
     ON = 0;
     dt = dtOff;
     break;
```

```
      //fprintf(outp1,"Walked off filament %d at vertex %d\n
         ", currentFil, currentV2);

     } else {
      vi--;
      currentV2 = filaments[currentFil][vi];
      //fprintf(outp1,"Changing direction. Now moving
         towards vertex %d\n", currentV2);
     }
    } else {
     if(currentV2 == filaments[currentFil][filaments[
        currentFil].size()-1]){
      OFF = 1;
      ON = 0;
      dt = dtOff;
      break;
      //fprintf(outp1,"Walked off filament %d at vertex %d\n
         ", currentFil, currentV2);

     } else {
      vi++;
      currentV2 = filaments[currentFil][vi];
      //fprintf(outp1,"Changing direction. Now moving
         towards vertex %d\n", currentV2);
     }
    }

  }
  // cargo moves towards the next vertex
  xc += dstep * cos(acos((xVertexPositions[currentV2]-xc)/(
     sqrt(pow((yVertexPositions[currentV2]-yc), 2) + pow((
     xVertexPositions[currentV2]-xc), 2)))));
  yc += dstep * sin(asin((yVertexPositions[currentV2]-yc)/(
     sqrt(pow((yVertexPositions[currentV2]-yc), 2) + pow((
     xVertexPositions[currentV2]-xc), 2)))));
  t += dt;
  //fprintf(outp1,"x: %lf\ty: %lf\tdt:%lf (On)\n", xc, yc,
     dt);
  //fprintf(outp1,"Distance to vertex %d: %lf (current
     filament: %d)\n", currentV2, sqrt(pow((yVertexPositions
     [currentV2]-yc), 2) + pow((xVertexPositions[currentV2]-
     xc), 2)), currentFil);


 }// end "on" movement
```

```
   // for storing cargo data in a file
   if (ON){
    fprintf(outp1, "%.3lf\t\t%.3lf\t\t%.3lf\t\t%.3lf\t\t%d\t%d
       \t\t%d\t\t\t%d\t\t\t%d\n", xc, yc, t, dt, ON,
       currentFil, BACKWARD, currentV2, i+1);
   }


  }// end movement of a single cargo


 }// end movement of all cargos
 fclose(outp1);
 switchProb += 0.1;
 }// end looping through switch probabilities

 return 0;
}
```

7.4.3 intersectionDistances.py

Calculates the average distance between filament intersections as well as the standard deviation of distances between filament intersections.

```
# intersectionDistances.py
# distance between intersections calculations
def main():

    # keep a list of distances between intersections
    intDists = []

    # open the file containing ditances between intersections
    with open('intersectionDistances.txt') as file:
        for line in file:
            intDists.append(float(line))

    # calculate the average distance between filament
       intersections
    avgIntDist = sum(intDists) / len(intDists)

    # check average
    print avgIntDist

    # calculate the variance
    intDistVar = 0.0
    for i in range(len(intDists)):
```

```
        intDistVar += (intDists[i] - avgIntDist) ** 2
    intDistVar /= len(intDists)

    # calculate the standard deviation
    intDistStdDev = (intDistVar) ** (0.5)

    # check the standard deviation
    print intDistStdDev



main()
```

7.4.4 analyzeDataRefined.py

Analyze the data after simulating the cargo movement.

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from matplotlib.lines import Line2D
import numpy as np

def squareFit(t,a):
    return a*t**(2.0)

def walkerMsdFit(t,a,b):
    # a is "effective" distance between filament intersections
    # b is "effetive" switching probability
    # "gamma" = (v / a) = (1 / a)
    # "alpha" = "beta" = cos(b*pi/2)
    return (1/((1/a)**2.0)/((1-(np.cos(b*np.pi/2))**2.0))
       **2.0) * \
           ((1-(np.cos(b*np.pi/2))**2.0)*(2+2*np.cos(b*np.pi
              /2))*((1/a)*t) - \
            2*((1+np.cos(b*np.pi/2))**2.0) + \
            np.exp(-(1/a)*t)*\
            (((2*np.cos(b*np.pi/2)+((np.cos(b*np.pi/2))**2.0)
               *(4+2*np.cos(b*np.pi/2))) / \
              (np.cos(b*np.pi/2)))* \
             (np.sinh(np.cos(b*np.pi/2)*(1/a)*t))+ \
             2*((1+np.cos(b*np.pi/2))**2.0)*(np.cosh(np.cos(b*
                np.pi/2)*(1/a)*t))))

def expFitPDF(t,a,b):
    return a*np.exp(-b*t)
```

```
def expFitCDF(t,a):
    return (1 - np.exp(-a*t))

def openFile(fileName):
    with open(fileName) as file:
        trajTemp = [[value for value in line.split()] for line
            in file]
    return trajTemp

def xytGet(trajTemp):
    # create dictionaries for the trajectories
    # keep track of x, y, and t values for each cargo number (
       trajectory)
    xTrajTemp = {}
    yTrajTemp = {}
    tTrajTemp = {}

    i = 1

    # check the trajectory file one line at a time
    while i < len(trajTemp):

        # the current cargo number (trajectory)
        cargoNumber = trajTemp[i][8]

        if cargoNumber in xTrajTemp:
            # add the x, y, t values to their corresponding
               key/list
            xTrajTemp[cargoNumber].append(float(trajTemp[i
               ][0]))
            yTrajTemp[cargoNumber].append(float(trajTemp[i
               ][1]))
            tTrajTemp[cargoNumber].append(float(trajTemp[i
               ][2]))

        else:
            # a new cargo number was found
            # create a new list
            xTrajTemp[cargoNumber] = [float(trajTemp[i][0])]
            yTrajTemp[cargoNumber] = [float(trajTemp[i][1])]
            tTrajTemp[cargoNumber] = [float(trajTemp[i][2])]

        # move to the next line in the file
        i += 1
```

```
    return xTrajTemp,yTrajTemp,tTrajTemp


def tortsGet(xDict,yDict,tDict):
    # find the tortuosity for each trajectory
    tortsTemp = {}
    tortCounts = 0.0
    tortSum = 0.0
    # go through each trajectory
    for key in tDict:
        x = xDict[key]
        y = yDict[key]
        i = 1
        pathLength = 0
        while i < len(tDict[key]):
            dx = x[i] - x[i-1]
            dy = y[i] - y[i-1]
            dis2 = dx*dx + dy*dy
            pathLength += (dis2)**(0.5)
            i += 1
        finalDis = ((x[len(xDict[key])-1]-x[0])**2.0 + (y[len(
           yDict[key])-1]-y[0])**2.0)**(0.5)
        tortsTemp[key] = [pathLength,finalDis]


    # check the tortuosity parameters and keep track of "large
       " tortuosity
    largesTemp = []
    for key in tortsTemp:
        if tortsTemp[key][1] == 0.0:
            tortsTemp[key] = 0
        else:
            tortsTemp[key] = tortsTemp[key][0] / tortsTemp[key
               ][1]

        tortCounts += 1.0
        tortSum += tortsTemp[key]


        # add cargo number to list of those with "large"
           tortuosities
        if tortsTemp[key] >= 2.0:
            largesTemp.append(key)

    avgTemp = tortSum / tortCounts
```

```
    return tortsTemp,largesTemp,avgTemp

def msdGet(xDict,yDict,tDict):
    # dictionary of MSDs for all trajectories
    # (will be a dictionary of lists)
    allMsdsTemp = {}


    # go through each trajectory
    for key in tDict:
        # x, y values for each cargo number (trajectory)
        x = xDict[key]
        y = yDict[key]
        # list of MSDs for each trajectory
        msdList = []
        # the current size of the time step "window"
        windowSize = 1
        # calculate msd for each window size
        while windowSize < len(tDict[key]):
            dis2tot = 0
            counts = 0
            i = windowSize
            # start moving the time step "window"
            while i < len(tDict[key]):
                dx = x[i] - x[i - windowSize]
                dy = y[i] - y[i - windowSize]
                dis2 = dx*dx + dy*dy
                dis2tot += dis2
                # how many times dis2 was calculated for this
                # time "window"
                counts += 1
                # move the window to the next x, y values
                i += 1
            # msd for the current time step window
            msd = dis2tot / counts
            msdList.append(msd)
            # increase the size of the "window" by one time
               step
            windowSize += 1

        # add the current MSD list to the dictionary of all
           MSDs
        allMsdsTemp[key] = msdList


    # calculate the ensemble average MSD
```

```
    msdListAvgTemp = []
    for i in range(1000):
        tot = 0
        counts = 0
        # go through each trajectory MSD
        for key in allMsdsTemp:
            if len(allMsdsTemp[key]) > i:
                tot += allMsdsTemp[key][i]
                # the number of trajectories that last up to
                   this time step
                counts += 1
        if counts != 0:
            avg = float(tot)/counts
            msdListAvgTemp.append(avg)


    return allMsdsTemp,msdListAvgTemp

def distributionsGet(xDict,yDict):
    # make a dictionary of run lengths, one for each cargo
    runLengths = {}
    maxRunLength = 0
    # calculate run lengths for all cargos
    for key in xDict:
        xList = xDict[key]
        yList = yDict[key]
        i = 1
        runLength = 0
        while i < len(xList):
            dx = xList[i] - xList[i-1]
            dy = yList[i] - yList[i-1]
            dis2 = dx*dx + dy*dy
            runLength += (dis2) ** (0.5)
            i += 1
        runLengths[key] = runLength
        if runLength > maxRunLength:
            maxRunLength = int(runLength)


    # calculating run length distribution for cargos
    lengthDistTemp = {}
    maxCount = 0
    for key in runLengths:
        lengthInt = int(runLengths[key])
        if lengthInt > maxCount:
            maxCount = lengthInt
```

```
            if lengthInt not in lengthDistTemp:
                lengthDistTemp[lengthInt] = 1
            else:
                lengthDistTemp[lengthInt] += 1

        lengthDistListTemp = [0] * (maxCount + 1)
        for i in range(len(lengthDistListTemp)):
            if i in lengthDistTemp:
                lengthDistListTemp[i] = lengthDistTemp[i]

        # normalize run length distribution
        totalCounts = 0
        for i in range(len(lengthDistListTemp)):
            totalCounts += lengthDistListTemp[i]

        for i in range(len(lengthDistListTemp)):
            lengthDistListTemp[i] /= float(totalCounts)

        # run length values
        runLengthValuesTemp = list(range(0,maxRunLength+1))
        runLengthValuesTemp = np.array(runLengthValuesTemp)


        # calculate run length distribution CDF
        lengthDistCDFTemp = []
        cdfCounter = 0
        for i in range(len(lengthDistListTemp)):
            lengthDistCDFTemp.append(cdfCounter)
            cdfCounter += lengthDistListTemp[i]

        # fit the PDF and CDF
        fitParamsPDFTemp,fitCovPDFTemp = curve_fit(expFitPDF,
           runLengthValuesTemp,lengthDistListTemp)
        fitParamsCDFTemp,fitCovCDFTemp = curve_fit(expFitCDF,
           runLengthValuesTemp,lengthDistCDFTemp)

        return runLengthValuesTemp,lengthDistListTemp,
           lengthDistCDFTemp,fitParamsPDFTemp,fitCovPDFTemp,
           fitParamsCDFTemp,fitCovCDFTemp

# reads and analyzes data from the multiple trajectories text
   file
def main():

    # time values used in MSD plots (each time step lasts 0.1
       s)
```

```
times = list(range(0,1001))
for i in range(len(times)):
    times[i] /= 10.0

times = np.array(times)

switchProbs =
   [0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

#print switchProbs


# open all files
traj00 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.000
   kOff0.001.txt')
traj01 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.100
   kOff0.001.txt')
traj02 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.200
   kOff0.001.txt')
traj03 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.300
   kOff0.001.txt')
traj04 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.400
   kOff0.001.txt')
traj05 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.500
   kOff0.001.txt')
traj06 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.600
   kOff0.001.txt')
traj07 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.700
   kOff0.001.txt')
traj08 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.800
   kOff0.001.txt')
traj09 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb0.900
   kOff0.001.txt')
traj10 = openFile('
   multipleCargoTrajectoriesNumCargos1000SwitchProb1.000
   kOff0.001.txt')
```

```
# the above is in the form traj[row][column]


# get x,y,t dictionaries
xTraj00,yTraj00,tTraj00 = xytGet(traj00)
xTraj01,yTraj01,tTraj01 = xytGet(traj01)
xTraj02,yTraj02,tTraj02 = xytGet(traj02)
xTraj03,yTraj03,tTraj03 = xytGet(traj03)
xTraj04,yTraj04,tTraj04 = xytGet(traj04)
xTraj05,yTraj05,tTraj05 = xytGet(traj05)
xTraj06,yTraj06,tTraj06 = xytGet(traj06)
xTraj07,yTraj07,tTraj07 = xytGet(traj07)
xTraj08,yTraj08,tTraj08 = xytGet(traj08)
xTraj09,yTraj09,tTraj09 = xytGet(traj09)
xTraj10,yTraj10,tTraj10 = xytGet(traj10)


# get all tortuosities and average tortuosity for each set
    of data
torts00,larges00,avgTort00 = tortsGet(xTraj00,yTraj00,
   tTraj00)
torts01,larges01,avgTort01 = tortsGet(xTraj01,yTraj01,
   tTraj01)
torts02,larges02,avgTort02 = tortsGet(xTraj02,yTraj02,
   tTraj02)
torts03,larges03,avgTort03 = tortsGet(xTraj03,yTraj03,
   tTraj03)
torts04,larges04,avgTort04 = tortsGet(xTraj04,yTraj04,
   tTraj04)
torts05,larges05,avgTort05 = tortsGet(xTraj05,yTraj05,
   tTraj05)
torts06,larges06,avgTort06 = tortsGet(xTraj06,yTraj06,
   tTraj06)
torts07,larges07,avgTort07 = tortsGet(xTraj07,yTraj07,
   tTraj07)
torts08,larges08,avgTort08 = tortsGet(xTraj08,yTraj08,
   tTraj08)
torts09,larges09,avgTort09 = tortsGet(xTraj09,yTraj09,
   tTraj09)
torts10,larges10,avgTort10 = tortsGet(xTraj10,yTraj10,
   tTraj10)

#print avgTort10

#tortMax = 0.0
#for key in torts10:
```

```
    #print torts10[key]
    #plt.plot(key,torts10[key],'o')
#plt.show()


# plot average tortuosity as a function of switching
   probability
#plt.scatter(0.0,avgTort00)
#plt.scatter(0.1,avgTort01)
#plt.scatter(0.2,avgTort02)
#plt.scatter(0.3,avgTort03)
#plt.scatter(0.4,avgTort04)
#plt.scatter(0.5,avgTort05)
#plt.scatter(0.6,avgTort06)
#plt.scatter(0.7,avgTort07)
#plt.scatter(0.8,avgTort08)
#plt.scatter(0.9,avgTort09)
#plt.scatter(1.0,avgTort10)
#plt.xlabel('switching probability')
#plt.ylabel('average tortuosity')
#plt.show()


#im = plt.imread('s5part1__cmle001.png')

#implot = plt.imshow(im,extent
   =[0,512*0.0675,512*0.0675,0])
#implot = plt.imshow(im,extent=[0,34.75,34.75,0])
#for key in tTraj00:
#   if key in larges00:
#         plt.plot(xTraj00[key],yTraj00[key],color='yellow
   ')
         #plt.annotate(key,xy=(xTraj00[key][0],yTraj00[key
            ][0]),color='yellow',size=8)
#   else:
#         plt.plot(xTraj00[key],yTraj00[key],color='blue')
        #plt.annotate(key,xy=(xTraj[key][0],yTraj[key][0])
            ,color='lime',size=8)

#plt.xlabel('x ($\mu m$)')
#plt.ylabel('y ($\mu m$)')
#plt.show()


# get all MSDs and the average MSD for each set of data
allMSDs00,msdListAvg00 = msdGet(xTraj00,yTraj00,tTraj00)
```

```
allMSDs01,msdListAvg01 = msdGet(xTraj01,yTraj01,tTraj01)
allMSDs02,msdListAvg02 = msdGet(xTraj02,yTraj02,tTraj02)
allMSDs03,msdListAvg03 = msdGet(xTraj03,yTraj03,tTraj03)
allMSDs04,msdListAvg04 = msdGet(xTraj04,yTraj04,tTraj04)
allMSDs05,msdListAvg05 = msdGet(xTraj05,yTraj05,tTraj05)
allMSDs06,msdListAvg06 = msdGet(xTraj06,yTraj06,tTraj06)
allMSDs07,msdListAvg07 = msdGet(xTraj07,yTraj07,tTraj07)
allMSDs08,msdListAvg08 = msdGet(xTraj08,yTraj08,tTraj08)
allMSDs09,msdListAvg09 = msdGet(xTraj09,yTraj09,tTraj09)
allMSDs10,msdListAvg10 = msdGet(xTraj10,yTraj10,tTraj10)

distList = []
probList = []

# finding fit for average MSDs
fitList00,tempList00 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg00[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList00[0])
probList.append(fitList00[1])

fitList01,tempList01 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg01[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList01[0])
probList.append(fitList01[1])

fitList02,tempList02 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg02[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList02[0])
probList.append(fitList02[1])

fitList03,tempList03 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg03[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList03[0])
probList.append(fitList03[1])

fitList04,tempList04 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg04[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList04[0])
probList.append(fitList04[1])

fitList05,tempList05 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg05[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList05[0])
probList.append(fitList05[1])
```

```
fitList06,tempList06 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg06[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList06[0])
probList.append(fitList06[1])

fitList07,tempList07 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg07[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList07[0])
probList.append(fitList07[1])

fitList08,tempList08 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg08[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList08[0])
probList.append(fitList08[1])

fitList09,tempList09 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg09[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList09[0])
probList.append(fitList09[1])

fitList10,tempList10 = curve_fit(walkerMsdFit,times[:100],
   msdListAvg10[:100],bounds=(0,[20.0,1.0]))
distList.append(fitList10[0])
probList.append(fitList10[1])

#print distList
#print probList

# plot distances between filament intersections and
   switching probabilities
# plot them in the same figure but using two axes
#plt.plot(switchProbs,distList,'o')
#plt.plot(switchProbs,probList,'o')
#plt.xlabel('switching probability (used in simulation)')
#plt.ylabel('effective distance between filament
   intersections')
#plt.ylabel('effective switching probability')

#figShared, ax1 = plt.subplots()
#ax2 = ax1.twinx()
#ax1.plot(switchProbs,distList,'o',color='blue')
#ax2.plot(switchProbs,probList,'o',color='green')
#ax1.set_xlabel('switching probability (used in simulation
   )')
#ax1.set_ylabel('effective distance between filament
   intersections',color='blue')
```

```
#ax2.set_ylabel('effective switching probability',color='
   green')


#plt.show()


# plotting all MSDs for each trajectory
#for key in allMSDs03:
#    plt.scatter(times[:len(allMSDs03[key])],allMSDs03[key
   ])


# plot the ensemble average MSD
plt.plot(times[:100],np.array(msdListAvg00[:100]),'o')
plt.plot(times[:100],walkerMsdFit(times[:100],fitList00
   [0],fitList00[1]),label='switch probability: 0.0')

plt.plot(times[:100],np.array(msdListAvg01[:100]),'o')
plt.plot(times[:100],walkerMsdFit(times[:100],fitList01
   [0],fitList01[1]),label='switch probability: 0.1')

plt.plot(times[:100],np.array(msdListAvg02[:100]),'o')
plt.plot(times[:100],walkerMsdFit(times[:100],fitList02
   [0],fitList02[1]),label='switch probability: 0.2')

#plt.plot(times[:100],np.array(msdListAvg03[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList03
   [0],fitList03[1]),label='switch probability: 0.3')

#plt.plot(times[:100],np.array(msdListAvg04[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList04
   [0],fitList04[1]),label='switch probability: 0.4')

#plt.plot(times[:100],np.array(msdListAvg05[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList05
   [0],fitList05[1]),label='switch probability: 0.5')

#plt.plot(times[:100],np.array(msdListAvg06[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList06
   [0],fitList06[1]),label='switch probability: 0.6')

#plt.plot(times[:100],np.array(msdListAvg07[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList07
   [0],fitList07[1]),label='switch probability: 0.7')
```

```
#plt.plot(times[:100],np.array(msdListAvg08[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList08
   [0],fitList08[1]),label='switch probability: 0.8')

#plt.plot(times[:100],np.array(msdListAvg09[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList09
   [0],fitList09[1]),label='switch probability: 0.9')

#plt.plot(times[:100],np.array(msdListAvg10[:100]),'o')
#plt.plot(times[:100],walkerMsdFit(times[:100],fitList10
   [0],fitList10[1]),label='switch probability: 1.0')



plt.xlabel('time (s)')
plt.ylabel('MSD $\mu m^2 / s$')
plt.legend(loc='upper left')
plt.show()

# get run length distribution information
runLengthValues00,lengthDistList00,lengthDistCDF00,
   fitParamsPDF00,fitCovPDF00,fitParamsCDF00,fitCovCDF00 =
    distributionsGet(xTraj00,yTraj00)
runLengthValues01,lengthDistList01,lengthDistCDF01,
   fitParamsPDF01,fitCovPDF01,fitParamsCDF01,fitCovCDF01 =
    distributionsGet(xTraj01,yTraj01)
runLengthValues02,lengthDistList02,lengthDistCDF02,
   fitParamsPDF02,fitCovPDF02,fitParamsCDF02,fitCovCDF02 =
    distributionsGet(xTraj02,yTraj02)
runLengthValues03,lengthDistList03,lengthDistCDF03,
   fitParamsPDF03,fitCovPDF03,fitParamsCDF03,fitCovCDF03 =
    distributionsGet(xTraj03,yTraj03)
runLengthValues04,lengthDistList04,lengthDistCDF04,
   fitParamsPDF04,fitCovPDF04,fitParamsCDF04,fitCovCDF04 =
    distributionsGet(xTraj04,yTraj04)
runLengthValues05,lengthDistList05,lengthDistCDF05,
   fitParamsPDF05,fitCovPDF05,fitParamsCDF05,fitCovCDF05 =
    distributionsGet(xTraj05,yTraj05)
runLengthValues06,lengthDistList06,lengthDistCDF06,
   fitParamsPDF06,fitCovPDF06,fitParamsCDF06,fitCovCDF06 =
    distributionsGet(xTraj06,yTraj06)
runLengthValues07,lengthDistList07,lengthDistCDF07,
   fitParamsPDF07,fitCovPDF07,fitParamsCDF07,fitCovCDF07 =
    distributionsGet(xTraj07,yTraj07)
```

```
    runLengthValues08,lengthDistList08,lengthDistCDF08,
       fitParamsPDF08,fitCovPDF08,fitParamsCDF08,fitCovCDF08 =
        distributionsGet(xTraj08,yTraj08)
    runLengthValues09,lengthDistList09,lengthDistCDF09,
       fitParamsPDF09,fitCovPDF09,fitParamsCDF09,fitCovCDF09 =
        distributionsGet(xTraj09,yTraj09)
    runLengthValues10,lengthDistList10,lengthDistCDF10,
       fitParamsPDF10,fitCovPDF10,fitParamsCDF10,fitCovCDF10 =
        distributionsGet(xTraj10,yTraj10)

    # plot run length distribution
    #plt.plot(runLengthValues,lengthDistList,'o',label='PDF')
    #plt.plot(runLengthValues,expFitPDF(runLengthValues,
       fitParamsPDF[0],fitParamsPDF[1]),label='PDF fit')
    #plt.plot(runLengthValues,lengthDistCDF,'o',label='CDF')
    #plt.plot(runLengthValues,expFitCDF(runLengthValues,
       fitParamsCDF[0]),label='CDF fit')
    #plt.xlabel('run length ($\mu$m)')
    #plt.ylabel('probability values')
    #plt.show()



main()
```

7.4.5 analyzeDataUMass2.py

Analyze the experimental data obtained through tracking the movement of cargos *in vitro* on a network of microtubule bundels.

```
import openpyxl
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import math
import numpy as np
import matplotlib

def expFitPDF(t,a,b):
    return a*np.exp(-b*t)

def expFitCDF(t,a,b):
    return (a/b)*(1 - np.exp(-b*t))

def main():
    #open and read from the two files
```

```
#1-motor tracks
wb1 = openpyxl.load_workbook('QD_1to1_tracks_for_Ajay.xlsx
   ')
#10-motor tracks
wb10 = openpyxl.load_workbook('QD_1to10_tracks_for_Ajay.
   xlsx')

#get necessary sheets
sheet1 = wb1.get_sheet_by_name('Sheet1')
sheet10 = wb10.get_sheet_by_name('Sheet1')

#keep a dictionary of TrackID keys
tracks1 = {}
tracks10 = {}

#building 1-motor track dictionary
for row in range(2,sheet1.max_row+1):
    trackId = int(sheet1['A'+str(row)].value)
    # convert x,y values to um
    x = float(sheet1['C'+str(row)].value)/1000
    y = float(sheet1['D'+str(row)].value)/1000
    t = sheet1['E'+str(row)].value

    #building a map to a list of lists
    if trackId not in tracks1:
        #if new trackId, start building lists
        x1 = [x]
        y1 = [y]
        t1 = [t]
        tracks1[trackId] = [x1,y1,t1]

    else:
        x1.append(x)
        y1.append(y)
        t1.append(t)
        tracks1[trackId] = [x1,y1,t1]

#building 10-motor track dictionary
for row in range(2,sheet10.max_row+1):
    trackId = int(sheet10['A'+str(row)].value)
    # convert x,y values to um
    x = float(sheet10['C'+str(row)].value)/1000
    y = float(sheet10['D'+str(row)].value)/1000
    t = sheet10['E'+str(row)].value

    #building a map to a list of lists
```

```
        if trackId not in tracks10:
            #if new trackId, start building lists
            x10 = [x]
            y10 = [y]
            t10 = [t]
            tracks10[trackId] = [x10,y10,t10]
        else:
            x10.append(x)
            y10.append(y)
            t10.append(t)
            tracks10[trackId] = [x10,y10,t10]

    # start making run length distribution calculations

    # make a dictionary of run lengths, one for each track
    runLengths1 = {}
    runLengths10 = {}

    # calculate run lengths for all one-motor tracks
    for key in tracks1:
        xList = tracks1[key][0]
        yList = tracks1[key][1]
        i = 1
        runLength = 0
        while i < len(xList):
            dx = xList[i] - xList[i-1]
            dy = yList[i] - yList[i-1]
            dis2 = dx*dx + dy*dy
            runLength += (dis2) ** (0.5)
            i += 1
        runLengths1[key] = runLength



    # calculate run lengths for all ten-motor tracks
    for key in tracks10:
        xList = tracks10[key][0]
        yList = tracks10[key][1]
        i = 1
        runLength = 0
        while i < len(xList):
            dx = xList[i] - xList[i-1]
            dy = yList[i] - yList[i-1]
            dis2 = dx*dx + dy*dy
            runLength += (dis2) ** (0.5)
            i += 1
```

```
        runLengths10[key] = runLength


    # calculating run length distribution for one-motor tracks
    lengthDist1 = {}
    maxCount = 0
    for key in runLengths1:
        lengthInt = int(runLengths1[key])
        if lengthInt > maxCount:
            maxCount = lengthInt
        if lengthInt not in lengthDist1:
            lengthDist1[lengthInt] = 1
        else:
            lengthDist1[lengthInt] += 1

    lengthDistList1 = [0] * (maxCount + 1)
    for i in range(len(lengthDistList1)):
        if i in lengthDist1:
            lengthDistList1[i] = lengthDist1[i]

    # calculating run length distribution for one-motor tracks
    lengthDist10 = {}
    maxCount = 0
    for key in runLengths10:
        lengthInt = int(runLengths10[key])
        if lengthInt > maxCount:
            maxCount = lengthInt
        if lengthInt not in lengthDist10:
            lengthDist10[lengthInt] = 1
        else:
            lengthDist10[lengthInt] += 1

    lengthDistList10 = [0] * (maxCount + 1)
    for i in range(len(lengthDistList10)):
        if i in lengthDist10:
            lengthDistList10[i] = lengthDist10[i]

    # testing
    #print lengthDist1
    #print lengthDistList1
    #print lengthDist10
    #print lengthDistList10

    # normalize run length distributions
    # one-motor tracks
    totalCounts = 0
```

```
for i in range(len(lengthDistList1)):
    totalCounts += lengthDistList1[i]

for i in range(len(lengthDistList1)):
    lengthDistList1[i] /= float(totalCounts)

# ten-motor tracks
totalCounts = 0
for i in range(len(lengthDistList10)):
    totalCounts += lengthDistList10[i]

for i in range(len(lengthDistList10)):
    lengthDistList10[i] /= float(totalCounts)

# run length values
runLengthValues = list(range(0,36))
runLengthValues = np.array(runLengthValues)
#print runLengthValues

# calculate the run length distribution CDFs
# one-motor tracks
lengthDistCDF1 = []
cdfCounter = 0
for i in range(len(lengthDistList1)):
    cdfCounter += lengthDistList1[i]
    lengthDistCDF1.append(cdfCounter)

# ten-motor tracks
lengthDistCDF10 = []
cdfCounter = 0
for i in range(len(lengthDistList10)):
    lengthDistCDF10.append(cdfCounter)
    cdfCounter += lengthDistList10[i]

# fit the CDFs
fitParamsCDF1,fitCovCDF1 = curve_fit(expFitCDF,
   runLengthValues,lengthDistCDF1[:36])
fitParamsCDF10,fitCovCDF10 = curve_fit(expFitCDF,
   runLengthValues[:34],lengthDistCDF10[:34])


print fitParamsCDF1, fitParamsCDF10

# plot run length distributions
#plt.plot(lengthDistList1[:36])
#plt.plot(lengthDistList10[:36])
```

```
plt.plot(lengthDistCDF1[:36],'o',label='one-motor tracks
   CDF')
plt.plot(runLengthValues,expFitCDF(runLengthValues,
   fitParamsCDF1[0],fitParamsCDF1[1]),label='one-motor
   tracks CDF fit')
plt.plot(lengthDistCDF10[:36],'o',label='ten-motor tracks
   CDF')
plt.plot(runLengthValues[:34],expFitCDF(runLengthValues
   [:34],fitParamsCDF10[0],fitParamsCDF10[1]),label='ten-
   motor tracks CDF fit')
plt.xlabel('run length ($\mu$m)')
plt.ylabel('CDF for run length distribution')
#plt.text(20,0.6,'one-motor CDF: $1 - e^{-0.208x}$')
#plt.text(20,0.5,'ten-motor CDF: $1 - e^{-0.142x}$')
plt.legend()
plt.show()

# start making MSD calculations

#lists of MSDs
msdList1 = []
msdList10 = []


#extract a trajectory list from the dictionary
for key in tracks1:
    xytv = tracks1[key]
    #start making a list of mean squared displacements
    passNum = 1
    msdList = []

    while passNum < len(xytv[2]):
        dis2tot = 0
        counts = 0
        i = passNum

        while i < len(xytv[2]):
            dx = xytv[0][i] - xytv[0][i-passNum]
            #print dx
            dy = xytv[1][i] - xytv[1][i-passNum]
            #print dy
            dis2 = dx*dx + dy*dy
            dis2tot = dis2tot + dis2
            counts = counts + 1
            i = i + 1
```

```
            msd = dis2tot / counts
            msdList.append(msd)
            passNum = passNum + 1

        msdList1.append(msdList)



#extract a trajectory list from the dictionary
for key in tracks10:
    xytv = tracks10[key]
    #start making a list of mean squared displacements
    passNum = 1
    msdList = []

    while passNum < len(xytv[2]):
        dis2tot = 0
        counts = 0
        i = passNum

        while i < len(xytv[2]):
            dx = xytv[0][i] - xytv[0][i-passNum]
            #print dx
            dy = xytv[1][i] - xytv[1][i-passNum]
            #print dy
            dis2 = dx*dx + dy*dy
            dis2tot = dis2tot + dis2
            counts = counts + 1
            i = i + 1

        msd = dis2tot / counts
        msdList.append(msd)
        passNum = passNum + 1

    msdList10.append(msdList)

#lists of average MSDs
msdList1Avg = []
msdList10Avg = []

#print msdList1[1]

msd1Num = [0]*120
msd10Num = [0]*160
```

```
# calculating MSD averages
for i in range(500):
    tot = 0
    counts = 0
    for item in msdList1:
        if len(item) > i:
            tot = tot + item[i]
            counts = counts + 1
            msd1Num[i] += 1
    if counts != 0:
        avg = float(tot)/counts
        msdList1Avg.append(avg)

for i in range(500):
    tot = 0
    counts = 0
    for item in msdList10:
        if len(item) > i:
            tot = tot + item[i]
            counts = counts + 1
            msd10Num[i] += 1
    if counts != 0:
        avg = float(tot)/counts
        msdList10Avg.append(avg)



# MSD plots
#fig = plt.figure()
#ax1 = fig.add_subplot(111)
#line1 = ax1.plot(msdList1Avg,label='1 motor MSD average')
#line2 = ax1.plot(msdList10Avg,label='10 motor MSD average
   ')
#ax1.set_xlabel('Number of time steps')
#ax1.set_ylabel('MSD $(nm^2)$')
#ax1.legend()
#ax2 = ax1.twinx()
#line3 = ax2.plot(msd1Num,'g',label='1-motor tracks')
#line4 = ax2.plot(msd10Num,'r',label='10-motor tracks')
#ax2.set_ylabel('Tracks remaining')
#ax2.legend(loc='center right')

# image plots
#im = plt.imread('QD_1to10_ch1005_mts.png')
#im = plt.imread('QDlom_1to1_1to50MT_MT_001.png')
```

```
    #implot = plt.imshow(im,extent
       =[0,512*0.0675,512*0.0675,0])
    #for key in tracks1:
    #    plt.plot(tracks1[key][0],tracks1[key][1],color='blue
       ')
    #    plt.annotate(key,xy=(xTraj[key][0],yTraj[key][0]),
       color='lime',size=8)


    #plt.show()

main()
```

Bibliography

[1] T. Soldati and M. Schliwa. Powering membrane traffic in endocytosis and recycling. *Nature Reviews: Molecular Cell Biology*, 7(12):897–908, 2006.

[2] D. Arcizet, B. Meier, E. Sackmann, J. O. Rädler, and D. Heinrich. Temporal analysis of active and passive transport in living cells. *Physical Review Letters*, 101(24):248103, 2008.

[3] P. C. Bressloff and J. M. Newby. *Stochastic models of intracellular transport*. Oxford Centre for Collaborative Applied Mathematics, 2013.

[4] S. M. Ali Tabei, S. Burov, H. Y. Kim, A. Kuznetov, T. Huynh, J. Jureller, L. H. Philipson, A. R. Dinner, and N. F. Sherer. Intracellular transport of insulin granules is a subordinated random walk. *Proceedings of the National Acedemy of Sciences USA*, 110(13):4911–4916, 2013.

[5] W. M. Saxton and P. J. Hollenbeck. The axonal transport of mitochondria. *Journal of Cell Science*, 118(Pt 23):5411–5419, 2012.

[6] J. L. Ross, M. Y. Ali, and D. M. Warshaw. Cargo transport: molecular motors navigate a complex cytoskeleton. *Current Opinion in Cell Biology*, 20(1):41–107, 2008.

[7] A. L. Wells, A. W. Lin, L. Q. Chen, D. Safer, S. M. Cain, T. Hasson, B. O. Carragher, R. A. Milligan, and H. L. Sweeney. Myosin vi is an actin-based motor that moves backwards. *Nature*, 401(6752):508–508, 1999.

[8] C. Loverdo, O. Bénichou, M. Moreau, and R. Voituriez. Enhanced reaction kinetics in biological cells. *Nature Physics*, 4(2):134–137, 2008.

[9] Z. Wang, S. Khan, and M. P. Sheetz. Single cytoplasmic dynein molecule movements: characterization and comparison with kinesin. *Biophysical Journal*, 69(5):2011–2023, 1995.

[10] G. M. Langford. Actin- and microtubule-dependent organelle motors: interrelationships between the two motility systems. *Current Opinion in Cell Biology*, 7(1):82–88, 1995.

[11] `http://www.i2bc.paris-saclay.fr/spip.php?article1136&lang=en`. Accessed: November 19, 2016.

[12] `http://micro.magnet.fsu.edu/cells/microtubules/microtubules.html`. Accessed: November 19, 2016.

[13] R. Mallik and S. P. Gross. Molecular motors: strategies to get along. *Current Biology*, 14(22):R971–R982, 2004.

[14] S. Klumpp, T. M. Nieuwenhuizen, and R. Lipowsky. Movements of molecular motors: ratchets, random walks and traffic phenomena. *Physica E*, 29(1–2):380–389, 2005.

[15] S. Klumpp and R. Lipowsk. Cooperative cargo transport by several molecular motors. *Proceedings of the National Academy of Sciences*, 102(48):17284–17289, 2005.

[16] J. Beeg, S. Klumpp, R. Dimova, R. S.Garcia, and E. Unger. Transport of beads by several kinesin motors. *Biophysical Journal*, 94(2):532–541, 2008.

[17] M. J. I. Müller, S. Klumpp, and R. Lipowsky. Tug-of-war as a cooperative mechanism for bidirectional cargo transport by molecular motors. *Proceedings of the National Academy of Sciences*, 105(12):4609–4614, 2007.

[18] D. Ando, M.K. Mattson, J. Xu, and A. Gopinathan. Cooperative protofilament switching emerges from inter-motor interference in multiple-motor transport. *Scientific reports*, 4, 2014.

[19] K.C. Huang, C. Vega, and A. Gopinathan. Conformational changes, diffusion and collective behavior in monomeric kinesin-based motility. *Journal of Physics: Condensed Matter*, 23(37):374106, 2011.

[20] J. Helenius, G. Brouhard, Y. Kalaidzidis, S. Diez, and J. Howard. The depolymerizing kinesin mcak uses lattic diffusion to rapidly target microtubule ends. *Nature*, 441(7089):115–119, 2006.

[21] W. H. Liang, Q. Li, K. M. R. Faysal, S. J. King, A. Gopinathan, and J. Xu. Microtubule defects influence kinesin-based transport in vitro. *Biophysical Journal*, 110(10):2229–2240, 2016.

[22] I. A. Telley, P. Bieling, and T. Surrey. Obstacles on the microtubule reduce the processivity of kinesin-1 in a minimal in vitro system and in cell extract. *Biophysical Journal*, 96(8):3341–3353, 2009.

[23] M. W. Gramlich, L. Conway, W. H. Liang, J. A. Labastide, S. J. King, J. Xu, and J. L. Ross. Single molecule investigation of kinesin-1 motility using engineered microtubule defects. *Scientific Reports*, 7(1):44290, 2017.

[24] L. Conway, M. W. Gramlich, S. M. Ali Tabei, and J. L. Ross. Microtubule orientation and spacing within bundles is critical for long-range kinesin-1 motility. *Cytoskeleton*, 71(11):595–610, 2014.

[25] I. Neri, N. Kern, and A. Parmeggiani. Modeling cytoskeletal traffic: An interplay between passive diffusion and active transport. *Physical Review Letters*, 110(9):098102, 2013.

[26] P. K. Trong, J. Guck, and R. E. Goldstein. Coupling of active motion and advection shapes intracellular cargo transport. *Physical Review Letters*, 109(2):028104, 2012.

[27] M. Scholz, S. Burov, K. L. Weirich, B. J. Scholz, S. M. Ali Tabei, M. L. Gardel, and A. R. Dinner. Cycling state that can lead to glassy dynamics in intracellular transport. *Physical Review X*, 6(1):011037, 2016.

[28] J. Snider, F. Lin, N. Zahedi, V. Rodlonov, C. C. Yu, and S. P. Gross. Intracellular actin-based transport: How far you go depends on how often you switch. *Proceedings of the National Academy of Sciences*, 101(36):13204–13209, 2004.

[29] Supravat Dey, Kevin Ching, and Moumita Das. Active and passive transport of cargo in a corrugated channel: A lattice model study. *The Journal of Chemical Physics*, 148(13):134907, 2018.

[30] Saurabh S. Mogre and Elena F. Koslover. Multimodal transport and dispersion of organelles in narrow tubular cells. *Phys. Rev. E*, 97:042402, Apr 2018.

[31] D. Ando, N. Korabel, K. C. Huang, and A. Gopinathan. Cytoskeletal network morphology regulates intracellular transport dynamics. *Biophysical Journal*, 109(8):1574–1582, 2015.

[32] A. Kahana, G. Kenan, M. Feingold, M. Elbaum, and R. Granek. Active transport on disordered microtubule networks: The generalized random velocity model. *Physical Review E*, 78(5 Pt 1):051912, 2008.

[33] Anne E Hafner and Heiko Rieger. Spatial organization of the cytoskeleton enhances cargo delivery to specific target areas on the plasma membrane of spherical cells. *Physical Biology*, 13(6):066003, 2016.

[34] Anne E. Hafner and Heiko Rieger. Spatial cytoskeleton organization supports targeted intracellular transport. *Biophysical Journal*, 114(6):1420 – 1432, 2018.

[35] E. Chevalier-Larsen and E. L. F. Holzbaur. Axonal transport and neurodegenerative disease. *BBA*, 1762(11–12):1094–1108, 2006.

[36] M. Aridor and L. A. Hannan. Traffic jam: A compendium of human diseases that affect intracellular transport processes. *Traffic*, 3(11):781–790, 2000.

[37] S. Condamin, O. Bénichou, V. Tejedor, R. Voituriez, and J. Klafter. First-passage times in complex scale-invariant media. *Nature*, 450(7166):77–80, 2007.

[38] T. M. Nieuwenhuizen, S. Klumpp, and R. Lipowsky. Walks of molecular motors interacting with immobilized filaments. *Physica A*, 350(1):122–130, 2005.

[39] B. M. Regner, D. Vučinić, C. Domnisoru, T. M. Bartol, M. W. Hetzer, D. M. Tartakovsky, and T. J. Sejnowski. Anomalous diffusion of single particles in cytoplasm. *Biophysical Journal*, 104(8):1652–1660, 2013.

[40] D. ben Avraham and S. Havlin. *Diffusion and Reactions in Fractals and Disordered Systems*. Cambridge University Press, 2000.

[41] J. H. Jeon, V. Tejedor, S. Burov, E. Barkai, C. Selhuber-Unkel, K. Berg-Sørenson, L. Oddershede, and R. Metzler. In vivo anomalous diffusion and weak ergodicity breaking of lipid granules. *Physical Review Letters*, 106(4):048103, 2011.

[42] A. E. Hafner and H. Rieger. Spatial cytoskeleton organization supports targeted intracellular transport. *Biophysical Journal*, 114(6):1420–1432, 2018.

[43] K. Schwarz, Y. Schröder, B. Qu, M. Hoth, and H. Rieger. Optimality of spatially inhomogeneous search strategies. *Physical Review Letters*, 117(6):068101, 2016.

[44] K. Chen, B. Wang, and S. Granick. Memoryless self-reinforcing directionality in endosomal active transport within living cells. *Nature Materials*, 14(6):589–593, 2015.

[45] A. G. Hendricks, E. Perlson, J. L. Ross, H. W. Schroeder III, M. Tokito, and E. L. F. Holzbaur. Motor coordination via tug-of-war mechanism drives bidirectional vesicle transport. *Current Biology*, 20(8):697–702, 2010.

[46] M. Weiss. Single-particle tracking data reveal anticorrelated fractional brownian motion in crowded fluids. *Physical Review E*, 88(1):010101, 2013.

[47] D. Cai, D. P. McEwen, J. R. Martens, E. Mayhöfer, and K. J. Verhey. Single molecule imaging reveals differences in microtubule track selection between kinesin motors. *PLOS Biology*, 7(10):e1000216, 2009.

[48] J. L. Ross. The impacts of molecular motor traffic jams. *Proceedings of the National Academy of Sciences of the United States of America*, 109(16):5911–5912, 2012.

[49] B. B. Mandelbrot and J. W. Van Ness. Fractional brownian motions, fractional noises and applications. *Society for Industrial and Applied Mathematics Review*, 10(4):422–437, 1968.

[50] R. Mietzler and J. Klafter. The random walk's guide to anomalous diffusion: A fractional dynamics approach. *Physics Reports*, 339(1):1–77, 2000.

[51] A. Caspi, R. Granek, and M. Elbaum. Diffusion and directed motion in cellular transport. *Physical Review E*, 66(1 Pt 1):011916, 2002.

[52] M. Dawson, D. Wirtz, and J. Hanes. Enhanced viscoelasticity of human cystic fibrotic sputum correlates with increasing microheterogeneity in particle transport. *Journal of Biological Chemistry*, 278(50):50393–50401, 2003.

[53] Z. Wang and D. C. Thurmond. Mechanisms of biphasic insulin-granule exocytosis – roles of the cytoskeleton, small GTPases and SNARE proteins. *Journal of Cell Science*, 122(Pt 7):893–903, 2009.

[54] Y. He, S. Burov, R. Metzler, and E. Barkai. Random time-scale invariant diffusion and transport coefficients. *Physical Review Letters*, 101(5):058101, 2008.

[55] R. T. Watson and J. E. Pessin. Glut4 translocation: The last 200 nanometers. *Cellular Signalling*, 19(11):2209–2217, 2007.

[56] A. Tomas, B. Yermen, L. Min, J. E. Pessin, and P. A. Halban. Regulation of pancreatic β-cell insulin secretion by actin cytoskeleton remodelling: role of gelsolin and cooperation with the mapk signalling pathway. *Journal of Cell Science*, 119(Pt 10):2156–2167, 2006.

[57] M. A. Kalwat and D. C. Thurmond. Signaling mechanisms of glucose-induced f-actin remodeling in pancreatic islet β cells. *Experimental and Molecular Medicine*, 45(8):e37, 2013.

[58] A. Gopinathan, K. C. Lee, J. M. Schwarz, and A. J. Liu. Branching, capping, and severing in dynamic actin structures. *Physical Review Letters*, 99(5):058103, 2007.

[59] A. E. Carlsson. Structure of autocatalytically branched actin solutions. *Physical Review Letters*, 92(23):238102, 2004.

[60] J. P. Bergman, M. J. Bovyn, F. F. Doval, A. Sharma, M. V. Gudheti, S. P. Gross, J. F. Allard, and M. D. Vershinin. Cargo navigation across 3d microtubule intersections. *PNAS*, 115(3):537–542, 2018.

[61] A. M. Stein, D. A. Vader, L. M. Jawerth, D. A. Weitz, and L. M. Sander. An algorithm for extracting the network geometry of three-dimensional collagen gells. *Journal of Microscopy*, 213(3):465–475, 2008.

[62] J. Taktikos. Modeling the random walk and chemotaxis of bacteria: Aspects of biofilm formation. *Dissertation*, 2012.

[63] A. Desai and T. J. Mitchison. Microtubule polymerization dynamics. *Annual Review of Cell and Developmental Biolgy*, 13(1):83–117, 1997.

[64] D. H. Schott, R. N. Collins, and A. Bretscher. Secretory vesicle transport velocity in living cells depends on the myosin-v lever arm length. *Journal of Cell Biology*, 156(1):35–39, 2002.