©Copyright 2012
Charles David Williams

A new view of the radial geometry in muscle: myofilament lattice spacing controls force production and energy storage.

Charles David Williams

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2012

Reading Committee:

Thomas L Daniel, Co-Chair

Michael Regnier, Co-Chair

Linda Wordeman

Program Authorized to Offer Degree:
Physiology and Biophysics

University of Washington

Abstract

A new view of the radial geometry in muscle:
myofilament lattice spacing controls
force production and energy storage.

Charles David Williams

Co-Chairs of the Supervisory Committee:
Professor Thomas L Daniel
Biology

Professor Michael Regnier
Bioengineering

Muscle is highly organized in both the axial direction and the radial direction, or the direction of contraction and the direction orthogonal to it. As muscle generates force, it does so in both the axial and radial directions. Lattice spacing, which is the radial spacing between its contractile filaments, increases as muscle shortens. Historically, the effects of these processes have not been accounted for in our conceptual or mathematical models of muscle contraction.

We develop a computational model of the half-sarcomere that is fully three dimensional and thus replicates the processes which occur in muscle's radial direction. This model employs a novel cross-bridge model which uses multiple springs, both extensional and angular. Where prior cross-bridge models use a change in rest length to generate force, our multi-spring model uses a lever arm mechanism similar to myosin's.

Using this model and experiments with isolated skinned muscle, we show that changes in lattice spacing increase the slope of the length tension curve by more than 20%. The length-tension curve describes the relationship between a muscle's sarcomere lengths and the maximum force which it can generate. The length-tension curve has been attributed to changing degrees of the overlap of thick and thin filaments as sarcomere length varies. A

steep slope on the length-tension curve is necessary for passive stability of muscular systems, such as cardiac muscle, which operate at short sarcomere lengths. These systems rely on the small length changes which accompany a new load to tailor the force produced for the new load. Without the steeper slope produced by varying lattice spacings, an external mechanism of force regulation would be necessary to provide system stability.

Additional model results show that substantial energy is stored in deformation of the cross-bridges during maximum activation, more than twice that which is stored in deformation of the thick and thin filaments. This energy is highly correlated with force produced in the radial direction, itself of the same order of magnitude as the axial force produced. These relative levels of axial and radial forces are themselves a confirmation of previous experimental measurements of radial force. The stored energy may play a role in powering rapid, one-off explosive movements such as prey striking by Mantis shrimp and tongue extension in toads.

TABLE OF CONTENTS

Page

List of Figures . . . iii

Glossary . . . v

Chapter 1: Introduction . . . 1
- 1.1 Muscle's structure . . . 1
- 1.2 A brief history of muscle models . . . 3
- 1.3 The action of a single cross-bridge . . . 4
- 1.4 The Length-Tension Relationship . . . 5
- 1.5 Energy storage for explosive movement . . . 7

Chapter 2: Axial and Radial Forces of Cross-bridges Depend on Lattice Spacing . . . 11
- 2.1 Abstract . . . 11
- 2.2 Author Summary . . . 12
- 2.3 Introduction . . . 12
- 2.4 Results . . . 14
- 2.5 Discussion . . . 18
- 2.6 Models . . . 21
- 2.7 Acknowledgments . . . 30

Chapter 3: The slope of the length-tension curve in muscle depends on lattice spacing. . . . 40
- 3.1 Abstract . . . 40
- 3.2 Introduction . . . 40
- 3.3 Results . . . 43
- 3.4 Discussion . . . 46
- 3.5 Methods . . . 47

Chapter 4: Elastic energy storage and radial forces in the myofilament lattice depend on sarcomere length . . . 60

4.1 Abstract . . . 60
4.2 Author Summary . . . 60
4.3 Introduction . . . 61
4.4 Results . . . 63
4.5 Discussion . . . 65
4.6 Methods . . . 67
4.7 Acknowledgements . . . 71

Chapter 5: Conclusions and Open Questions . . . 79
5.1 Conclusions . . . 79
5.2 Open Questions . . . 81

Bibliography . . . 85

Appendix A: Appendix A . . . 94
A.1 File crossbridge.py . . . 94

Appendix B: Appendix B . . . 104
B.1 File af.py . . . 104
B.2 File mf.py . . . 116
B.3 File mh.py . . . 129
B.4 File hs.py . . . 144
B.5 File run.py . . . 163

Appendix C: Appendix C . . . 168
C.1 File aws.py . . . 168

LIST OF FIGURES

Figure Number | Page

1.1 Axial and radial directions 8
1.2 Vertebrate and insect lattices 9
1.3 Constant volume contraction 10

2.1 Cross-bridge types and kinetic scheme 31
2.2 Forces, energy, and kinetics of the XB models at rest lattice spacing 32
2.3 Energy and kinetics of the multi-spring cross-bridges 33
2.4 Post-power stroke forces of the 2sXB and 4sXB models 34
2.5 Axial and radial post-power stroke forces as separate components 35
2.6 Changes in step size with lattice spacing 35
2.7 Changes in cross-bridge resting geometry with the power stroke 36
2.8 Single cross-bridge simulation protocol 37
2.9 Cross-bridge free energy and kinetics at multiple lattice spacings 38

3.1 A spatially explicit three-dimensional multifilament sarcomere model 53
3.2 Isovolumetric, isolattice, and isolength force curves 54
3.3 Model force at all lattice spacings and sarcomere lengths 55
3.4 Normalized force of skinned fibers under osmotic compression 56
3.5 Multi-filament model architecture and geometry 57
3.6 Skinned fiber bundle apparatus 58
3.7 Lattice arrangement 59

4.1 Axial and radial force orientations 72
4.2 Example axial and radial forces 73
4.3 Radial and axial forces are of the same order of magnitude 73
4.4 Filament and cross-bridge energy partitioning 74
4.5 Contractile lattice energies versus sarcomere length 75
4.6 Thick, thin, extensional, and torsional energy partitioning 75
4.7 Model code structure 76
4.8 AWS sequence diagram 77

5.1 Oil droplet deformation under radial force 84

GLOSSARY

AXIAL DIRECTION: parallel to the direction of muscle shortening. Aligned with the thick and thin filaments.

RADIAL DIRECTION: perpendicular to the direction of muscle shortening. Orthogonal to the thick and thin filaments.

LATTICE SPACING: the distance between thick and thin filaments. Unless otherwise specified, we give lattice spacing as the distance between the surfaces of adjacent thick and thin filaments, rather than the distance from the center of one to the center of another.

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to the Daniel Lab, the Regnier Lab, the University of Washington, and all the little gremlins contained therein. The author's stalwart friends and family are due a hearty thanks, as well.

DEDICATION

to dear Ariah

Chapter 1

INTRODUCTION

Muscle is a highly ordered tissue whose properties we rely on yet don't fully understand. Muscle contraction powers locomotion, flight, and the pumping of blood by the heart. Over the last hundred years, muscle's operation has been predominantly characterized by a few simple relationships such as that between muscle's length and the tension it can generate [Rassier et al., 1999, Hill, 1938, Gordon et al., 1966]. Our thinking about these relationships has been primarily concerned with how the muscle's constituent proteins react to changes in the axial direction that occur with changes in muscle length (see Figure 1.1 for an orientation to axial and radial directions). However, this fails to take into account a large cause of changes to the environment in which muscle's myosin motors operate, radial expansion and contraction. Here, through a combination of theoretical models and experiments, we look at the role that changes in the radial direction play in controlling the generation of force and muscle energetics.

1.1 Muscle's structure

Muscle is composed of successive bundles of parallel structures. Whole skeletal muscles are bounded by a layer of collagenous connective tissue, the epimysium, Within the epimysium are bundles of muscle cells or fibers. These bundles of fibers, fascicles, are bound by the perimysium, another layer of collagenous connective tissue. Each fiber contains multiple myofibrils, long chains of contractile machinery. These myofibrils are composed of a series of sarcomeres laid end-to-end. Sarcomeres are an interdigitating lattice of contractile protein filaments; these filaments are described according to their size and composition.

The sarcomere contains two main types of contractile filaments: thick and thin. The thick filaments are composed primarily of myosin, a motor protein which changes confirmation as it interacts with the thin filament to generate force. The thin filaments are

composed primarily of a double-helical coiled-coil of actin and its associated regulatory proteins. These thin filament regulatory proteins allow and control the binding of myosin to actin, and thus the levels of force which muscle generates [Gordon et al., 2000]. The regulatory proteins, Troponin and Tropomyosin, allow myosin binding when shifted from a closed state to a permissive state with elevated Ca^{2+} levels or through mechanical perturbation. The thin filaments stretch towards the center of the sarcomere from either end, where they are anchored in the z-disks which separate adjacent sarcomeres from each other. The thick filaments, in contrast, occupy the center of the sarcomere and stretch towards either end. In the region where the thick and thin filaments overlap, myosin may bind to actin, change shape, and thus generate force. The degree to which one set of filaments overlaps the other is determined by their lengths and the overall length of the sarcomere. Filament lengths change little, as the filaments' architecture makes them quite stiff, while overall sarcomere length changes proportionate to muscle length [Suzuki and Sugi, 1983, Rassier et al., 1999].

1.1.1 Geometry of the contractile lattice

The geometry of the sarcomere described so far is aligned primarily in the axial direction, the direction in which muscle shortens, shown relative to a thick filament in Figure 1.1. However, the sarcomere is also highly organized in the radial direction, orthogonal to the axial direction (again shown in Figure 1.1). Sarcomeres are an interdigitating lattice of thick and thin filaments; in the region in which the filaments overlap and generate force through their interactions, the relative arrangement of thick and thin filaments is very well described by a regular pattern. The lattice of thick and thin filaments varies from organism to organism but is based on a hexagonal lattice of thick filaments in both vertebrate striated muscle and insect flight muscle, as shown in Figure 1.2A [Millman, 1998]. These two systems differ in the arrangement of the thin filaments around the thick filaments. In vertebrate muscle the thin filaments are located so that they are accessed by and equidistant from three surrounding thick filaments. In insect flight muscle the thin filaments are located equidistant between two thick filaments. This difference means that while vertebrate muscle has a ratio of thick to thin filaments of 1:3, insect flight muscle has a ratio of 1:2. A greater number

of thick filaments accessing a single thin filament increases the possibility of cooperative activation of the thin filament via cross-bridge formation, a means increasing the Ca^{2+} sensitivity of the system. When comparing results from the insect and vertebrate systems, my modeling and experimental work avoids any effects of differences in Ca^{2+} sensitivity between them by exploring only maximum force values at full activation.

1.2 A brief history of muscle models

Muscle models, based on both underlying mechanisms of action and on mathematical descriptions of the properties of bulk muscle tissue, have long been used to express our understanding of force generation. A.V. Hill's work began the process of mathematically describing muscle's action [Hill, 1938]. While this was a quantitative model of force generation that well describes the development of force upon stimulation, it is not based upon the mechanisms underlying force generation. The first predictive models of muscle contraction, which derived muscle properties from the properties of lower levels of muscle's structure, began with A.F. Huxley's work in the 1950s [Huxley, 1957]. Huxley described the cross-bridges which generate force in muscle as a population with a fraction in each of several states. In a well-mixed model, determination of the fraction of cross-bridges in each state is governed by the external conditions imposed on the muscle. This explained muscle's properties as consequences of the properties of its constituent parts but, in treating the cross-bridges as a homogeneous population, did not account for interactions between cross-bridges linked by the contractile lattice. More recently, spatially explicit models represented the thick and thin filaments as chains of springs that could be linked by cross-bridges, also represented as springs [Daniel et al., 1998, Tanner et al., 2007, Campbell, 2009]. These spatially explicit models allowed a bound cross-bridge to affect the binding and kinetics of neighboring cross-bridges, first through movement of thick and thin filaments and later through coupled activation of adjacent thin filament regulatory proteins. Spatially explicit models thus included the internal interactions of adjacent and separated cross-bridges within the sarcomere, a previously unidentified mechanism by which the fraction of bound and force generating cross-bridges is regulated. We build upon prior spatially explicit models to create a multi-filament model which is sensitive to processes in the radial direction (Figure 1.2B).

1.3 The action of a single cross-bridge

The spatially explicit models and the well-mixed models that preceded them both treated a cross-bridge as a single spring in the axial direction. This single spring cross-bridge, like myosin itself, binds and then undergoes a power-stroke that generates force. The power-stroke of this single spring cross-bridge, however, is simulated by a change in the rest length of the spring. This is in contrast to the known mechanism by which myosin actually generates force. As long ago as 1965, Reedy et al.'s electron micrographs of insect flight muscle demonstrated that cross-bridges are rotated by approximately 45 between their rest and force generating states [Reedy et al., 1965]. This force generation by rotation about a pivot point came to be known as the lever arm mechanism, as it uses a section of the myosin head as a lever by which it amplifies the angle change into a substantial movement in the axial direction [Huxley, 1969]. Long and complex efforts to obtain crystallographic structures of the myosin head were successful in the early 1990s and showed the existence of four distinct regions: 1) where myosin attaches to actin , 2) about which myosin pivots (the converter domain), 3) which act as a movement amplifying lever arm (the light chain domain), 4) and which transmit this force to the thick filament backbone [Rayment et al., 1993, Spudich, 2001]. This structure offers a template for the design of an alternative model of the cross-bridge, a task described in Chapter 2.

One major limitation of the single spring cross-bridge model prompts the creation of a simulated cross-bridge based on the lever arm mechanism: the single-spring model exists in a single dimension. Because the single spring cross-bridge is collinear in the axial direction, it is insensitive to changes in lattice spacing and the production of forces not aligned with the direction of shortening. Both of these limitations alter how we interpret basic properties of muscle such as the length-tension relationship and energy storage.

A multi-spring model of the cross-bridge can use a lever arm mechanism to generate force. The lever arm is represented as an angular spring which changes its rest angle to generate force, this necessitates a rotation about the converter domain and thus moves the cross-bridge in both the axial and radial directions. Movement in both axial and radial dimensions means the geometry of the multi-spring cross-bridge responds to changes in

lattice spacing, the consequences of which we discuss in Chapter 3. Changes in cross-bridge angle also produce forces in the radial direction, which we discuss in Chapter 4.

Why haven't prior computational models of the cross-bridge used a lever arm mechanism? Single spring cross-bridge models have been used in spatially explicit models because they limit computational complexity. The force of a single spring is described by a linear equation and so allows the model's internal forces to be balanced through matrix operations. The non-linear components of the multi-spring model require the use of slower iterative solution techniques. However, the computational resources now available through remote cloud-based processing have advanced to a point where the solution methods required by a multi-spring model are not time-prohibitive.

1.4 The Length-Tension Relationship

The length-tension (LT) relationship is a fundamental property of striated muscle. As muscle's length changes, it is able to generate differing levels of force. As muscle lengthens, the force it can generate on maximum activation is initially low, climbs to a peak, and then descends. The shape of the LT curve functions as a passive control mechanism for some muscle systems, such as in the case of the Frank-Starling law of the heart [Smith et al., 2009]. The Frank-Starling law describes cardiac muscle's increased force in response to perturbation by increased filling of the heart, a means of maintaining equilibrium volume. Cardiac muscle operates on the steep ascending limb of the LT curve where in a small incremental amount of stretching results in a greater than incremental extra force being generated. This regulates the strength of contraction in cardiac muscle where, as more blood fills the heart, cardiac muscle is stretched to a greater length, and thus more force is generated to eject the increased inflow of blood. This is a passive method of regulation. In cases such as these, the LT curve describes how muscle is able to maintain a stable system without active neural sensing and modulation. But while the importance of this regulation mechanism is well established, there is a flaw in our current understanding of its origin: we only consider events in the axial direction.

1.4.1 Classic Interpretation of the LT Curve

Knowledge of the LT relationship dates back to work done in the late 1800s, but our current interpretation was largely formed by Gordon, Huxley, and Julien's work in the 1960s [Blix, 1894, Gordon et al., 1966]. The classic interpretation of the origin of the LT curve is based on changes which occur in the axial direction, but muscle changes in more than the axial direction as a result of shortening, the radial geometry changes due to the constant volume relationship. As cell length shortens with contraction, it also increases in diameter. This, in turn, increases myofilament lattice spacing.

The LT curve has three major sections: the ascending limb, the plateau region, and the descending limb. Each of these phases is associated with a change in the number of cross-bridges able to form and generate force. At the long sarcomere lengths of the descending limb, there is little overlap between the thick and thin filaments. Because there are few myosin heads located opposite a thin filament, there are few cross-bridges which may form and generate force. As the sarcomere shortens, force increases. This rise in force is attributed to an increased overlap of thick and thin filaments and a corresponding increase in the number of heads that face and may bind to a thin filament.

At the long sarcomere lengths of the descending limb, there is little overlap between the thick and thin filaments. Because there are few myosin heads located opposite a thin filament, there are few cross-bridges which may form and generate force. As the sarcomere shortens, force increases. This rise in force is attributed to overlap increases and a corresponding increase in the number of heads that face and may bind to a thin filament.

As the sarcomere continues to shorten, all myosin heads eventually face an opposing thin filament and the sarcomere enters the plateau region of the LT curve. As all potential cross-bridges now have a chance to be active, the greatest amount of force is generated.

Shortening beyond the plateau region, thin filaments from opposite z-disks begin to overlap and sarcomere force declines. This decreased force marks the beginning of the ascending limb of the LT curve. The overlapping of thin filaments is believed to shield binding sites from facing myosin heads, leading to the decreased formation of cross-bridges and decreased force. At extremely short sarcomere lengths, thin filaments begin to crash

into the opposing sarcomere ends and force production drops off dramatically.

1.4.2 Muscle maintains a constant volume

When the length of muscle changes, its volume does not. This is the constant volume condition. This constant volume condition means that the spacing between the lattice spacing between the thick and thin filaments increases as muscle shortens, as shown in Figure 1.3 [Millman, 1998]. The constant volume condition means that as sarcomeres are shortening down the ascending limb, not only are thin filaments increasing in overlap with each other, the distance across which myosin must diffuse in order to bind is also increasing. Conversely, as muscle lengthens along the descending limb, lattice spacing is shrinking and a given cross-bridge becomes restricted to a smaller axial region of the opposing thin filament. Exploration of these processes is the subject of Chapter 3.

1.5 Energy storage for explosive movement

Storage of the energy generated by muscle through elastic deformation of materials such as tendon or cuticle has long been recognized as a means to realize explosive movement [Alexander and Bennet-Clark, 1977, Gosline et al., 2002]. Explosive movements involve quick and non-successive motion, such as the tentacular strike of the squid, tongue projection by toads and chameleons, or closing of the trap-jaw ant's mandibles [Kier and Leeuwen, 1997, de Groot and Leeuwen, 2004, Lappin et al., 2006, Gronenberg, 1996]. In these one-off movements, energy storage serves a different role than in the cyclical movements involved in terrestrial and airborne locomotion [Alexander, 1988, 2005]. In cyclical movements, energy storage is a means to reduce the metabolic requirements of locomotion. Energy storage in explosive movements serves as a mechanism which increases the instantaneous power output of the system by storing energy produced over a period of time longer than that of the motion. In both cyclical and explosive cases, most efforts to characterize the methods of energy storage have focused on accessory collagenous or otherwise resilient tissue such as tendons and cuticle [Roberts and Azizi, 2011]. In Chapter 4 we propose that muscle, particularly the cross-bridges, stores energy useful in explosive movements.

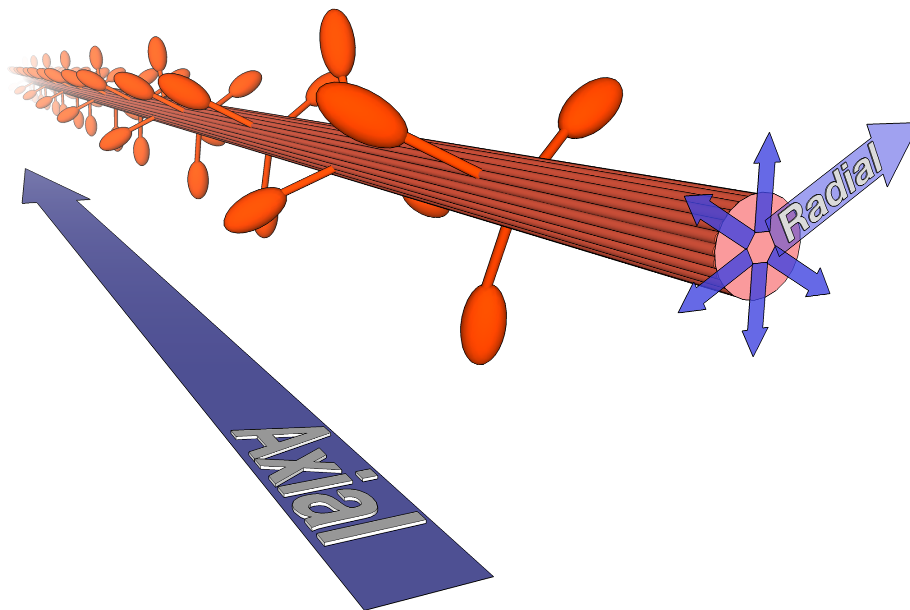


Figure 1.1: **Axial and radial directions in muscle are defined relative to the axes of the thick and thin filaments.** The axial direction is parallel to the long axis of the contractile filaments, and to the direction of muscle shortening. Orthogonal or perpendicular to the axial direction is the radial direction. Sarcomere length is measured in the axial direction, while lattice spacing is measured in the radial direction.

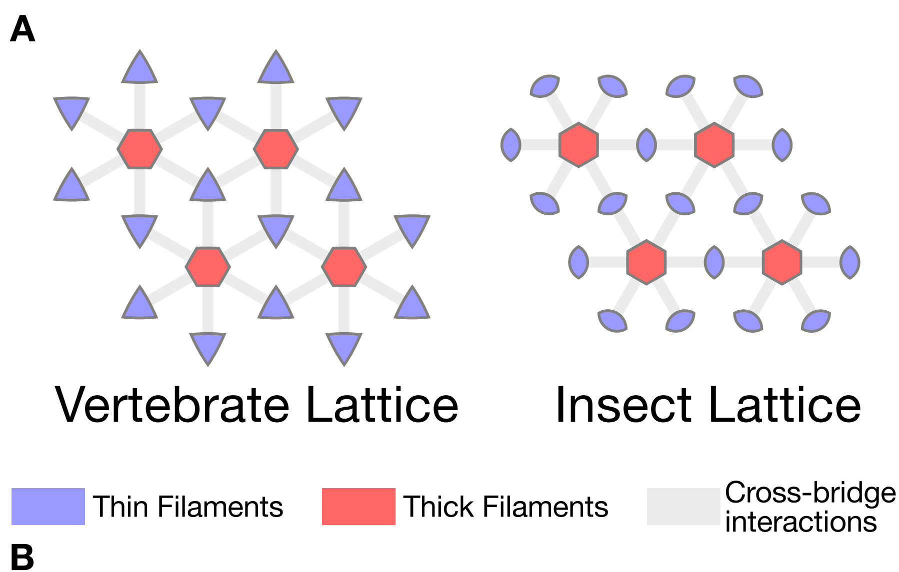


Figure 1.2: **Thick and thin filaments are regularly arranged in the contractile lattice of muscle. A)** Vertebrate and insect muscle lattice structures have differing thin filament arrangements. In both lattice types the thick filaments are arrayed in a hexagonal pattern. In vertebrate striated muscle, the thin filaments are each connected to three thick filaments, while in insect flight muscle, each thin filament faces only two thick filaments. **B)** Our multi-filament model consists of four thick and eight thin filaments arranged as in vertebrate striated muscle. Reflections across boundaries allows a limited number of filaments to simulate a semi-infinite lattice.

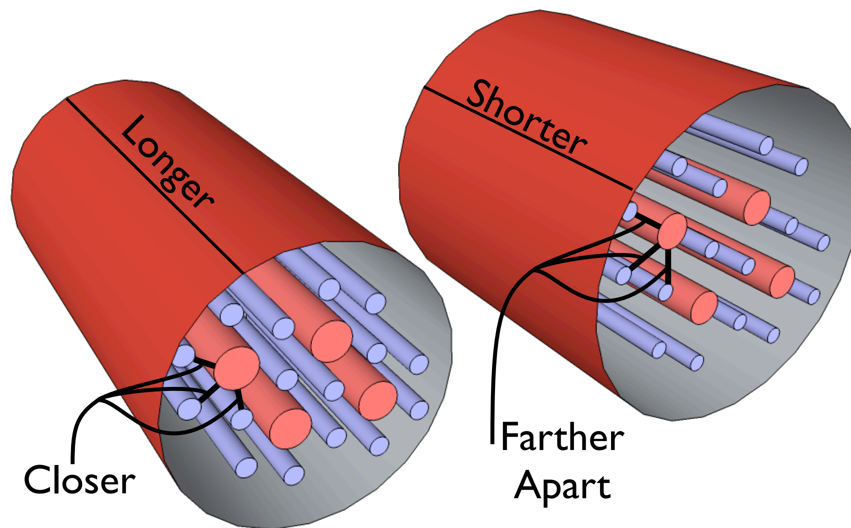


Figure 1.3: **Maintaing a constant volume during shortening leads to varying lattice spacing.** When muscle shortens while obeying the constant volume constraint, lattice spacing grows. To maintain a constant lattice volume, lattice spacing changes as the square root of 1/sarcomere length.

Chapter 2

AXIAL AND RADIAL FORCES OF CROSS-BRIDGES DEPEND ON LATTICE SPACING

C. David Williams, Michael Regnier, and Thomas L. Daniel

2.1 Abstract

Nearly all mechanochemical models of the cross-bridge treat myosin as a simple linear spring arranged parallel to the contractile filaments. These single-spring models cannot account for the radial force that muscle generates (orthogonal to the long axis of the myofilaments) or the effects of changes in filament lattice spacing. We describe a more complex myosin cross-bridge model that uses multiple springs to replicate myosin's force-generating power stroke and account for the effects of lattice spacing and radial force. The four springs which comprise this model (the 4sXB) correspond to the mechanically relevant portions of myosin's structure. As occurs *in vivo*, the 4sXB's state-transition kinetics and force production dynamics vary with lattice spacing.

Additionally, we describe a simpler two spring cross-bridge (2sXB) model which produces results similar to those of the 4sXB model. Unlike the 4sXB model, the 2sXB model requires no iterative techniques, making it more computationally efficient. The rate at which both multi-spring cross-bridges bind and generate force decreases as lattice spacing grows. The axial force generated by each cross-bridge as it undergoes a power stroke increases as lattice spacing grows. The radial force that a cross-bridge produces as it undergoes a power stroke varies from expansive to compressive as lattice spacing increases. Importantly, these results mirror those for intact, contracting muscle force production.

Keywords: cross-bridge model; cross-bridge kinetics; lattice spacing; radial force; muscle model; myosin

2.2 Author Summary

The molecular motor myosin drives the contraction of muscle, but doesn't just produce force in the axis of shortening. Models of muscle contraction have primarily treated myosin as a simple spring oriented parallel to its direction of movement. This assumption does not allow prediction of the relationship between the forces produced and the spacing between contractile filaments or of radial forces, perpendicular to the axis of shortening, all of which are observed during muscle contraction. We develop an alternative model, still computationally efficient enough to be used in simulations of the sarcomere, that incorporates both extensional and torsional (angle dependent, like those found in a watch) springs. Our model captures much of the spacing dependent kinetics and forces that are missing from single-spring models of the cross-bridge.

2.3 Introduction

Radial forces are the same order of magnitude as axial forces in contracting muscles [Maughan and Godt, 1981a, Cecchi et al., 1990, Millman, 1998]. These forces, along with axial force acting in the direction of muscle contraction, depend on myofilament lattice spacing [Bagni et al., 1994, Fuchs and Martyn, 2005]. At the same time, structural information about myosin cross-bridges suggests that they generate force by applying torque to a lever arm [Rayment et al., 1993, Uyeda et al., 1996, Huxley, 2000]. This lever arm generates the strain accompanying the power stroke via a change in the rest angle at which the lever is attached to S1 region [Huxley, 2000, Houdusse and Sweeney, 2001]. This change in angle occurs at the converter region, a flexible area in myosin S1 which acts as a torsional spring. These phenomena may be related: the radial forces a cross-bridge creates are results of the lever arm geometry (as suggested by Schoenberg [Schoenberg, 1980b]).

Existing theoretical and computational models of cross-bridge force generation at the level of the half-sarcomere assume that force is generated by a simple extensional linear spring oriented parallel to the long axis of the myofilaments (Figure 2.1A). This assumption has persisted from the earliest fundamental models of muscle contraction to more elaborate and spatially explicit models [Huxley, 1957, Daniel et al., 1998, Chase et al., 2004, Tanner

et al., 2007, Campbell, 2009]. These single-spring models yielded insight into the processes that regulate production of force in the direction of contraction, parallel to the long axis of the myofilaments. However, these prior models of muscle contraction have paid less attention to radial forces and the effects of changes in filament lattice spacing. As a result, geometries of the single spring cross-bridge models have changed little while kinetic schemes governing transitions between conformational states have increased in complexity [Huxley, 1957, Pate and Cooke, 1989, Daniel et al., 1998, Smith et al., 2008]. To analyze the radial forces that occur during muscle contraction, a different cross-bridge geometry is needed: a geometry that produces both forces aligned with and forces orthogonal to the long axis of the myofilaments. A lever arm of several springs can: (1) simulate the deformations a cross-bridge undergoes as it generates force through the power stroke, (2) provide a geometry which is practical for use in cross-bridge models, and (3) account for both axial and radial forces [Houdusse and Sweeney, 2001].

Here we detail two models of cross-bridges that use multiple springs to replicate the lever arm mechanism and capture its biologically relevant effects (Figure 2.1B-C). Both models are affected by changes in lattice spacing as well as axial offset from binding sites along the thin filament, and both account for the radial component of force produced during the power stroke. The first model (referred to as the 4sXB model) simulates the cross-bridge as a system of four linearly elastic springs arranged in a geometry based upon the structure of the S1 and S2 regions of myosin II (Figure 2.1C). Our second model (referred to as the 2sXB model) consists of two linearly elastic springs and provides greater computational efficiency than the 4sXB model while replicating many of the more complex model's behaviors (Figure 2.1B). A prior two spring cross-bridge model was proposed by Schoenberg (1980), with the S2 arm represented as an extensional spring and the S2-S1 junction as a torsional spring [Schoenberg, 1980a,b]. Both the 4sXB model and the 2sXB model use a three-state model of cross-bridge cycling kinetics, consisting of an unbound state, a low-force pre-power stroke state, and a force-producing post-power stroke state. The kinetics of transition from one state to another in our models are similar to those used previously but are generalized for use in two dimensions; our kinetics calculate transition probabilities using the free energy landscape of the cross-bridges instead of the offset of the cross-bridge head (Figures 2.1D

and 2.8) [Pate and Cooke, 1989, Daniel et al., 1998, Takagi et al., 2004, Tanner et al., 2007]. We compare the 4sXB and 2sXB models to a single spring model of the cross-bridge (referred to as the 1sXB model), similar to those used previously. We quantify both the axial and the radial forces of our two cross-bridge models. Additionally, we show how changes in lattice spacing and axial offset affect kinetics and forces in our multiple-spring models.

2.4 Results

The 4sXB and 2sXB models detailed here were developed to discover the consequences of lattice spacing on cross-bridge kinetics and two dimensional force production. Multi-spring cross-bridges introduce a lattice spacing dependence into force production and kinetics, and account for radial forces. As lattice spacing changes, the kinetics and forces of the 4sXB and 2sXB models shift in both magnitude and axial offset.

At 34 nm d_{10}, the multi- and single-spring cross-bridges have similar kinetics and energies At rest lattice spacing, the free energies and kinetics of the of the single- and multi-spring cross-bridge models are largely similar, as seen in Figure 2.2 (where the 1sXB values used are calculated as in Fig 10 of Tanner et al. (2007) [Tanner et al., 2007]). These properties share a common base that is intentionally conserved, where possible, between the multiple-spring and single-spring cross-bridges [Pate and Cooke, 1989]. The free energies of the multi-spring cross-bridges are a result of both extensional springs that are at an angle to the thick filament and torsional springs sensitive to the angle they make with the thick filament. As the multi-spring cross-bridges move in the axial direction, their angles to the thick filament backbone change. This angle dependence skews the free energies of the multi-spring cross-bridges from the symmetric hyperbola of the 1sXB (Figure 2.2A). The two-dimensional diffusion-based binding probability function that governs the multi-spring cross-bridges (as described in the binding rate calculation section) causes the likely binding areas to occupy a greater range of axial positions than those of the single-spring cross-bridge (Figure 2.2B) [Berg, 1993, Dill and Bromberg, 2003]. Multi-spring cross-bridges are thus less likely than the 1sXB model to bind near their rest position, but are more likely to bind than the 1sXB at greater offsets from their rest position. This flattening and spreading of

the binding probability function is a result of the extra degrees of freedom of motion in the two-dimensional models. The power stroke rate constants of the multi-spring cross-bridges are the same as those of the single-spring cross-bridge, with energy-dependent terms using the sum of the free energy of every spring comprising a cross-bridge (Figure 2.2C). The detachment rate constant of the 1sXB explicitly relies on cross-bridge head position as well as energy. This position dependence was removed in adapting the 1sXB model's detachment rate constant for the multi-spring cross-bridges. The detachment rate constant thus loses the intentional asymmetry that the position term provided and retains only the asymmetry created by the spring geometries of the 2sXB and 4sXB models (Figure 2.2D). The rate of detachment and the other cross-bridge kinetic rate constants remain close to those of the 1sXB, even though the kinetics of the multi-spring cross-bridges are based not on axial position but on the free energy of the cross-bridge in multiple dimensions.

Axial offsets of most cross-bridge properties decrease as lattice spacing grows
The axial offset of a cross-bridge property is the axial distance from the point where the cross-bridge attaches to the thick filament to the point where the cross-bridge property reaches an extreme value or inflection point. These axial offsets are depicted in Figures 2.3 and S2 where, for example, the axial offset of the 2sXB attachment rate constant at 34 nm d_{10} is approximately 12 nm. As lattice spacing increases, the axial offsets of most multi-spring cross-bridge kinetic rates and free energies grows smaller. This relationship is shown in Figures 2.3A and B and S2 A and B, where the axial offset of the 4sXB or 2sXB model's lowest energy point is more than 3 nm greater at a lattice spacing of 32 nm d_{10} than at a lattice spacing of 38 nm d_{10}. The positions where cross-bridges are most likely to bind shift to smaller axial offsets at larger lattice spacings, decreasing how extended a cross-bridge is likely to be upon binding (Figures 2.3C-D and 2.9C-D). Similarly, as lattice spacing increases, decreases in the axial offset of the power stroke rate constant inflection point cause the size of the power stroke to change with lattice spacing (Figures 2.3E-F and 2.9E-F). The 4sXB model's rate of detachment is the only cross-bridge property whose axial offset is predominately invariant with changes in lattice spacing (Figures 2.3G and 2.9G). This exception is explained by the largely radially aligned post-power stroke orientation of γ,

the 4sXB model's final spring. Combined, these effects reduce the axial force a cross-bridge generates at larger lattice spacings with implications for the sarcomere length dependence of force production and relaxation. These multi-spring cross-bridge models are the first to be capable of reproducing these lattice spacing dependent effects on force production and kinetics.

Probability of a cross-bridge being bound decreases as lattice spacing diverges from rest The number of cross-bridges in a force generating state depends on lattice spacing. At any axial location, as lattice spacing diverges from its 34 nm d_{10} rest value, the rate of attachment decreases while the rate of detachment increases (Figure 2.3C-D and 2.3G-H). These kinetic rate constants change with lattice spacing because they depend on the difference in free energy between the unbound state and the pre- or post-power stroke state, a difference which increases with lattice spacing. This increase in energy makes a cross-bridge increasingly likely to transition to the unbound state and remain there (Figure 2.3C-D and 2.3G-H). An example of the decrease in the likelihood of a cross-bridge remaining bound can be seen in the 4sXB model, where the slowest rate of detachment is 20/sec at a lattice spacing of 34 nm d_{10} but rises to 260/sec at 38 nm d_{10} (Figure 2.3G). As a result of these changes, individual cross-bridges spend less time in a bound state and are less likely to generate force as lattice spacing diverges from its rest value.

Forces at a given axial offset increase with lattice spacing The axial and radial forces at a given axial offset correlate with lattice spacing (Figures 2.4 and 2.5). When lattice spacing is compressed, more expansive radial forces and smaller axial forces are produced. When lattice spacing is expanded, more compressive radial forces and larger axial forces are produced. An example of increased forces with increased lattice spacing is seen in the 4sXB model which, at a 10 nm axial offset, produces half the radial and half the axial force at 35 nm d_{10} as it does at 38 nm d_{10} (Figure 2.5A-B). Similarly with the 2sXB model at a 12 nm axial offset, a lattice spacing of 35 nm d_{10} produces two thirds of the axial and radial forces as does a lattice spacing of 38 nm d_{10} (Figure 2.5C-D). At large lattice spacings, this greater force per cross-bridge competes with the decreased probability a cross-bridge will

bind and generate force, an interaction that requires a model of the half-sarcomere using multi-spring cross-bridges to fully evaluate [Martyn et al., 2004].

The force landscapes of Figure 2.5 also show that no lattice spacing is free of radial force at all axial offsets. The radial force produced by a cross-bridge, even at rest lattice spacing, increases in magnitude as the cross-bridge tip moves away from its unstrained axial offset.

Step size varies with lattice spacing The step size of both multi-spring models varies with lattice spacing (Figure 2.6). We define step size at a given lattice spacing as the axial distance between the pre- and post-power stroke positions of the myosin head. Put another way, step size at one lattice spacing is the distance from the axial offset with the lowest free energy in the pre-power stroke state, to the axial offset with the least amount of energy in the post-power stroke state. Both models have a peak step size at a relatively uncompressed lattice spacing, with decreasing step size as lattice spacing diverges from that value. The 4sXB model has a maximum step size of 5.0nm near 34nm lattice spacing and the 2sXB model has a maximum step size of 6.1nm near 36nm lattice spacing.

Radial forces are of the same order of magnitude as axial forces The radial and axial components of force, produced by a 4sXB model or 2sXB model moved from its rest position to an axial offset, are of the same order of magnitude (Figures 2.2E-F and 2.4A-D). The values of the axial and radial forces produced by the multiple-spring cross-bridge models at rest lattice spacing are compared to those produced by the single-spring cross-bridge model in Figure 2.2E-F. The relative values of the radial and axial forces are visualized as the angles of the force vectors in Figure 2.4A-D. Axial locations and lattice spacings with balanced axial and radial forces produce force vectors which are neither vertical nor horizontal, but in some intermediate orientation. Most axial and radial offsets are populated by such vectors, particularly regions a cross-bridge would be most likely to occupy (unlikely regions are not shown in the vector plots). The few regions dominated by one force, notably some small offset positions in the 2sXB model (Figure 2.4D), are dominated by radial forces. This presence of large radial forces suggests that, in all but the least strained locations at the smallest axial offsets, radial forces will be present in magnitudes comparable to those

of axial forces.

2.5 Discussion

Our multi-spring cross-bridge models show how myofilament lattice spacing influences cross-bridge properties, from axial and radial forces to kinetics and step size. The 4sXB and 2sXB models show two key features that differ significantly from prior models: (1) the inclusion of torsional springs and lever-arm mechanisms reveals a dependence of step size on lattice spacing and (2) this lever-arm mechanism produces radial forces and axial forces of the same magnitude, a ratio similar to that observed experimentally [Maughan and Godt, 1981a, Cecchi et al., 1990, Brenner and Yu, 1991a]. The dependencies of step size, force production, and kinetics on lattice spacing help explain measured changes in force generation with changes in lattice spacing [Millman, 1998].

Force generated by a multi-spring cross-bridge depends on lattice spacing The lattice spacing of the filaments around an attached multi-spring cross-bridge determine the energy landscape of the cross-bridge and thus the force it can generate. The forces and strains a cross-bridge produces at most axial offsets grow more positive as lattice spacing increases (Figure 2.4E-H). While this increased cross-bridge strain translates into greater axial and radial force per post-power stroke cross-bridge, the probability that these cross-bridges will bind decreases as lattice spacing increases (Figure 2.3C-D). The decrease in attachment rate constants at extreme lattice spacings, while power stroke rate constants remain unchanged (Figure 2.3E-F), suggests lattice spacing influences muscle fiber force generation by altering the rate of cross-bridge attachment rather than the power stroke rate [Martyn et al., 2004]. Spatially explicit effects in the compliant sarcomere, such as cross-bridge induced realignment of binding sites, may act to balance the decreased binding and increased detachment at larger lattice spacings.

The 2sXB model approximates the 4sXB model The energies, kinetics, and forces generated by the 2sXB model are subject to the same governing trends as those of the 4sXB model, and can be made similar by deliberate parameter choice (Table 2.1 and Figures 2.2,

2.3, 2.4, and 2.5). That the 2sXB model can replicate the results of the 4sXB model indicates two things: first, the 2sXB can be used in place of the 4sXB in larger simulations, enabling work that would otherwise require prohibitive resources, and second, a feature shared between our two models is responsible for the interesting properties of our simulations, the use of a lever arm which undergoes an angle change to generate force. While the energies, binding rate constants, and power stroke rate constants of the multi-spring cross-bridges are almost identical, there are some smaller differences between the two models. The rate constant of detachment is rotated by approximately 20° between the two systems due to differences in the way the post-power stroke position is achieved (Figure 2.3). The 4sXB model and the 2sXB model generate somewhat different forces; the axial force produced by each model increases with lattice spacing, but that produced by the 4sXB does so more steeply (Figure 2.5A, C). In a reversal of this pattern, the 2sXB model's radial force is more dependent on lattice spacing (Figure 2.5B, D). In each of these cases, the forces generated by both multi-spring cross-bridges are subject to the same trend. The close agreement between the forces and other properties of the two cross-bridge representations supports the position that the key feature of our multi-spring models is the use of a lever arm to generate force, rather than a factor unique to the 4sXB model, such as the simulation of interaction between the lever arm and the S2 domain. Substituting the 2sXB model for the 4sXB model reduces the runtime of a simulation by two orders of magnitude and puts multi-spring cross-bridge simulations of the half-sarcomere within reach.

Cross-bridge step size depends on lattice spacing, influences shortening velocity

The geometries of the multi-spring models require a change in step size accompany a change in lattice spacing. This is because, while the length of the lever arm changes as lattice spacing varies, the pre- and post-power stroke angles do not. Step size varies more in the 4sXB model as the 4sXB model's spring configuration causes the pre- and post-power stroke free energies to differ more than in the 2sXB model. As the detachment rate constant is a product of the post-power stroke free energy, the greater rotation in the 4sXB's post-power stroke free energy, relative to that of the 2sXB model, can be seen in Figure 2.3 G-H. Experimental measurements of step size vary, and it has been postulated that this is due

to more than experimental error, but to our knowledge these results are the first prediction of a step size that varies with lattice spacing [Brenner, 2006]. Experimental confirmation of these predictions is not possible with current literature: existing *in vivo* measurements of step size are from isolated myosin preparations which are unable to simulate a change in muscle lattice spacing [Howard, 2001, Peterman et al., 2004].

While our single cross-bridge models lack the predictive power of a multi-filament model, the dependence of step size on lattice spacing offers insight into unloaded shortening velocity. Maximum unloaded shortening velocity is commonly interpreted as a function of both myosin's step size and drag from attached post-power stroke cross-bridges [Gordon et al., 2000]. A decrease in unloaded shortening velocity is observed when lattice spacing is compressed via dextran [Goldman, 1987, Metzger and Moss, 1987]. This slower unloaded shortening is supported by the multi-spring models: their step size exhibits a similar decrease as lattice spacing shrinks (Figure 2.6). However, a moderate increase in the rate of detachment at highly compressed lattice spacings, seen in Figure 2.3 G–H, may balance smaller steps sizes. This increased detachment rate is due to the greater post-power stroke strain that is present with greater radial displacement of the cross-bridge. Changes in modeled detachment rates and step size are both likely to be needed, along with changes in filament overlap, to explain the complicated dependence of unloaded shortening velocity on sarcomere length [Edman, 1979].

Large radial component of forces may influence lattice spacing in multi-filament models The 4sXB and the 2sXB produce radial forces of the same order of magnitude as the axial forces generated by a cross-bridge. These forces range between 10% and 50% of the axial force at the least strained axial and radial offsets where a cross-bridge is most likely to enter the post-power stroke state (Figure 2.4). Muscle fibers display these radial forces by resisting width changes as osmotic pressure is applied [Maughan and Godt, 1981a]. Direct measurement of lattice spacing by X-ray diffraction has confirmed fiber width estimates of radial force [Matsubara et al., 1984]. Checchi et al. (1990) [Cecchi et al., 1990] observed large radial forces by examining lattice spacing during redevelopment of tension following length changes. A spatially explicit model, even one using multiple thick and thin filaments

arranged in a lattice, is insensitive to lattice spacing if it uses a version of the 1sXB model. Embedding multi-spring cross-bridges in a multi-filament model allows the simulation of radial force regulation in a lattice of thick and thin filaments. The inclusion of radial forces in a multi-filament model permits examination of previously unavailable kinds of cooperativity, ones where radial force can be transmitted through the backbone lattice to affect the kinetics of other cross-bridges. Radial force is a potential regulator of lattice spacing and of Ca^{2+} sensitivity as lattice spacing and sarcomere length vary [Millman, 1998]. A multi-filament model using the 4sXB or 2sXB can simulate the interaction of radial force generated by a cross-bridge with radial forces provided by other mechanisms, e.g. titin or electrostatic repulsion [Martyn et al., 2004, Cazorla et al., 2001, Millman, 1998]. Thus multi-spring cross-bridges make it possible to evaluate the influence of these radial forces, posited to be regulators of lattice spacing, and processes which may depend on lattice spacing or myosin head to thin filament distance, such as the Frank-Starling mechanism; something not possible with a 1sXB model [Smith et al., 2009].

In future studies, these models will permit the investigation of radial forces and lattice spacing in multi-filament models, and will allow us to examine disease states that alter myosin compliance. The inclusion of radial forces and lattice spacing in half-sarcomere models will illuminate regulatory mechanisms of shortening velocity and length-dependent axial force generation. Other efforts may use existing studies of how disease-related mutations alter myosin compliance to produce disease state mimicking cross-bridge models [Seebohm et al., 2009]. Multi-filament simulations using these altered cross-bridge models have the potential to explain how symptoms of disease states such as hypertrophic cardiomyopathy arise from myosin-level changes.

2.6 Models

Our two cross-bridge models, the 4sXB model and the 2sXB model (Figure 2.1B-C), are designed to capture a range of mechanical behaviors observed or posited by prior work, namely radial force generation and the effects of lattice spacing on cross-bridge binding and force generation. Both cross-bridge models are an arrangement of linearly elastic torsional (angular or watch-like) or Hookean (extensional) springs.

2.6.1 Geometry

Spring configurations To enable comparison with previous cross-bridge models, we implement a one-dimensional model in addition to our multi-spring models. This one-dimensional model uses a linearly elastic spring oriented parallel to the long axis of the thick filament (Figures 2.1A and 2.7A). The resulting cross-bridge forces are restricted to the direction of shortening, that is, axially oriented. The one-dimensional 1sXB model cannot yield radial forces. Moreover, this model's geometry is unable to account for changes in kinetics or forces at varying lattice spacing. This reference model is identical to those used in recent spatially-explicit computational analyses [Daniel et al., 1998, Chase et al., 2004, Tanner et al., 2007].

The 4sXB model uses two extensional and two torsional springs to represent the myosin head (Figures 2.1C and 2.7C). This arrangement of four springs corresponds closely to regions of the cross-bridge believed to regulate and respond to strain or deformation [Houdusse and Sweeney, 2001, Köhler et al., 2002]. In particular, the four springs correspond to the point where the S2 region attaches to the rod, the S2 region, the point where the S2 region attaches to the light chain domain (LCD), and the LCD. These points are labeled α, β, δ, and γ, respectively (Figure 2.1C). Rest values, stiffnesses, and their sources are detailed in Table 2.1.

The rest angle of δ decreases to simulate the transition from a pre-power stroke to a post-power stroke state (Figure 2.7C). This method of force generation acts in two dimensions and thus allows lattice spacing to influence forces and state transition rates. In the 4sXB, a change in the rest angle of δ mimics myosin's lever-arm mechanism of force generation [Houdusse et al., 2000, Houdusse and Sweeney, 2001]. As the extensional spring γ does not bend and the angle at which the globular domain attaches to actin remains unchanged, applying torque at one end of γ is equivalent to applying the opposite torque at the opposite end. Thus a change in the rest angle of δ produces a torque equivalent to that which the converter domain applies to the LCD during the power stroke.

The 2sXB model is a simplification of the 4sXB model, using one extensional spring (ρ) and one torsional spring (θ) to represent the myosin head (Figures 2.1B and 2.7B). The

2sXB treats the power stroke as a change in the rest angle of θ (Figure 2.7B); like the 4sXB, the 2sXB generates force by applying torque to a lever arm. The parameters of the 2sXB are set so that the pre- and post-power stroke tip location and kinetics of the 2sXB match those of the 4sXB model. In addition to the change in the rest angle of θ during the power stroke, we adjust the length of ρ so that the base-to-tip distance of the 2sXB in both the pre- and post-power stroke states is equal to the same measurement in the 4sXB model. The result is computationally simpler than the 4sXB model, but retains the 4sXB's two-dimensional behavior.

The 2sXB presented here is contrasted to an alternative geometry used by Schoenberg [Schoenberg, 1980a,b], where an extensional spring representing the S2 domain is joined, via a torsional spring, to a rigid rod representing the S1 domain. The use of this alternative geometry requires the position of the torsional spring linking the S1 and S2 domains be found through iterative solution methods whenever the cross-bridge tip position changes. This use of iterative solution methods is similar to that required by our 4sXB and imposes similarly large computational requirements when incorporated into larger spatially explicit models. Additionally, this alternative geometry restrains the cross-bridge tip to an area within one S1 length of the line in which the S2 segment is set.

Parameters used in both cross-bridge models are derived, where possible, from existing experimental data, described below. Each extensional spring (one in the 1sXB model, two in the 4sXB model and one in the 2sXB model) has a rest length and a spring constant, while each torsional spring (two in the 4sXB model and one in the 2sXB model) has a rest angle and a spring constant. The lengths and angles of the springs used for the 4sXB are based on tomographic reconstructions of *in vivo* S2 lengths and x-ray crystallographic reconstructions of the S1 fragment [Taylor et al., 1999, Rayment et al., 1993]. The rest length and angle of the springs used in the 2sXB model are set so that the tips of both the 2sXB's and 4sXB's simulated myosin heads are in the same location before and after the power stroke.

Calculation of lattice spacing The multi-spring models use an internal representation of lattice spacing that is analogous to the *in vivo* distance from the surface of a thick filament

to the surface of an adjacent thin filament. However, since this surface-to-surface lattice spacing (ssLS) is not commonly reported, we present lattice spacing as the d_{10} measurement used in x-ray diffraction studies of muscle [Millman, 1998]. The d_{10} value is the distance between the centers of mass of adjacent thick filaments. We calculate the d_{10} value that corresponds to a given ssLS using both the geometry of the cross-bridge and the lattice spacing at which the cross-bridge generates the least radial force. Specifically, d_{10} is found from $d_{10} = 1.5(ssLS + cf)$ [Millman, 1998]. The correction factor (cf) compensates for the filament radii: the difference between the ssLS surface-based measurement and the d_{10} center-of-mass-based measurement. The correction factor offset also sets the relationship between ssLS and d_{10} so that, at rest lattice spacing, the post-power stroke cross-bridge generates neither compressive nor tensile radial force. This offset becomes 6.90 nm when the rest d_{10} spacing is 34 nm [Brenner and Yu, 1991a]. The ssLS that correspond to the d_{10} spacings of interest are then calculated and define the window of lattice spacings we examine [Millman, 1998]. Thus the lattice spacing within the model is bound by experimental lattice spacings and is a function of both the geometry of the actomyosin lattice and the lattice spacing at which radial forces are minimized.

Displacement and force generation Each cross-bridge undergoes a distortion as myosin hydrolyzes ATP to ADP.P$_i$; this distortion is the basis of the power stroke [Pate and Cooke, 1989, Daniel et al., 1998, Tanner et al., 2007]. The energy liberated by the hydrolysis of ATP drives force generation by inducing strain in the cross-bridge, appearing as a change in the cross-bridge rest length [Howard, 2001]. For the 1sXB model, this distortion is represented as a change in the rest length of the cross-bridge's only spring (Figure 2.7A). The 4sXB and 2sXB models use a process which adheres more closely to the *in vivo* lever-arm mechanism; they represent the power stroke as a change in the rest angle of a torsional spring (Figures 2.7B,C) [Reedy, 2000]. The force generated by this process has both axial and radial components. The axial component of the force vector is the portion that lies along the long axes of the thick and thin filaments. The radial component of this vector lies perpendicular to the thick and thin filaments, orthogonal to the axial component. The relative values of the post-power stroke axial and radial forces are determined by the con-

struction of the cross-bridge (number of springs and their geometry), and the displacement of the cross-bridge tip from its rest position.

Calculation of spring lengths and angles To calculate the force and energy a cross-bridge produces and stores as its tip is displaced, we need to know the lengths and angles of the springs that constitute the cross-bridge. When the 1sXB model is placed under strain, the tip of its myosin head moves to a new axial offset. Finding the length of the 1sXB model's spring is simple, as it must span the complete distance from the cross-bridge tip to the thick filament attachment site. Finding the lengths and angles of springs in the 4sXB and 2sXB models is a two-dimensional problem; they must account for both the axial and radial distance from cross-bridge tip to cross-bridge base. The values of the 2sXB model's springs are determined analytically, as both spring values are set by the choice of a head location. The 2sXB model's spring values, ρ and θ, are given by $\rho(t_x, t_y) = (t_x^2 + t_y^2)^{1/2}$ and $\theta(t_x, t_y) = \arctan(t_y/t_x)$, for a cross-bridge tip location of (t_x, t_y) (Figure 2.1B). The 4sXB model has a greater number of springs and thus another point whose location must be defined: (δ_x, δ_y), the S2/LCD linking point where the angular spring δ is located (Figure 2.1C). The coordinates of the δ spring cannot be analytically determined, they must be found through iterative optimization. We use a modification of Powell's "dog-leg" method (from the SciPy computational package [Jones et al., 2001]) to locate the δ spring such that the 4sXB model is at its lowest energy state for the current cross-bridge tip position. Once δ's location is known, its angle, the angle of α and the lengths of β and γ are determined analytically. The angles and lengths for a given tip location (t_x, t_y) and δ location (δ_x, δ_y) are given by:

$$\begin{aligned}
\alpha(\delta_x, \delta_y) &= \arctan(\delta_y/\delta_x) \\
\beta(\delta_x, \delta_y) &= (\delta_x^2 + \delta_y^2)^{1/2} \\
\delta(\delta_x, \delta_y, t_x, t_y) &= \arctan((t_y - \delta_y)/(t_x - \delta_x)) + \pi - \alpha(\delta_x, \delta_y) \\
\gamma(\delta_x, \delta_y, t_x, t_y) &= ((t_x - \delta_x)^2 + (t_y - \delta_y)^2)^{1/2}
\end{aligned}$$

2.6.2 Kinetics

To describe the kinetics we use a simplified three-state model of the cross-bridge cycle originally described by Pate and Cooke (1989) [Pate and Cooke, 1989] and modified by Tanner et al. (2007) [Tanner et al., 2007]. This relatively simple scheme directly links the cross-bridge's kinetics and mechanics; the three kinetic states are directly comparable to the myosin configurations described in Houdusse (2000) [Houdusse et al., 2000]. The kinetic rates are independent of the number of springs used in a model cross-bridge, allowing the 4sXB and the 2sXB models to use the same system. The three states represented in the kinetic scheme are (1) an unbound state: Myosin-ADP.P_i (2) a loosely-bound state:Actin-Myosin-ADP.P_i and (3) a force-generating post-power stroke state: Actin-Myosin-ADP (Figure 2.1D). These kinetics replicate those of a generic cross-bridge, and are aimed at reproducing properties shared between cardiac, skeletal, and insect myosin types.

The kinetics of both the 4sXB and the 2sXB models are strain dependent and are essentially transforms of the free energy landscapes experienced by the cross-bridges in their different states. These free energies are a function of the distortion necessary to move the point representing the simulated myosin head's tip to the proposed binding site. Examples of these free energy landscapes are shown in Figure 2.3A and B, with cuts through them at the rest lattice spacing visible in Figure 2.2A. As the free energies of the cross-bridges are functions of their spring rest values and stiffnesses, changing the geometry and stiffness of the springs used by the model also changes the kinetics of the model.

The binding probabilities of both the 4sXB and the 2sXB models are determined by Monte-Carlo simulations of their diffusion as a result of being perturbed by Boltzmann-derived energy distributions [Dill and Bromberg, 2003]. After a new head location is found, a binding probability is calculated which decreases exponentially with distance from the potential binding site. This probability is tested against a random number from a uniform distribution to determine if binding occurs in our chosen time step of 1 ms.

Free energy in each state The total free energy liberated by the hydrolysis of the gamma P_i of ATP and available to the myosin head over the course of a cross-bridge cycle

(ΔG) depends on both the standard free energy of ATP hydrolysis ($\Delta G_{0,\text{ATP}}$) and the concentrations of ATP, ADP and P_i. The free energy available to the cross-bridge over its cycle is given by $\Delta G = -\Delta G_{0,\text{ATP}} - \ln \frac{[\text{ATP}]}{[\text{ADP}][\text{P}_i]}$. The free energy in the unbound state serves as a reference for the other states and is set to 0. As the unbound cross-bridge supports no strain, its free energy (U_1) remains at 0 for all axial offsets and lattice spacings. Only a portion of the liberated free energy is available to the cross-bridge in a given state. The limits on available ΔG are included in the free energy of each state as an efficiency factor, as in Tanner et al. (2007) [Pate and Cooke, 1989, Tanner et al., 2007] . The weakly bound state's efficiency is 28%, represented with $\alpha_e = 0.28$, and the strongly bound state's efficiency is 68%, represented with $\eta_e = 0.68$. The free energy of a cross-bridge in each state also depends on the strain the cross-bridge experiences from distortion upon binding. Thus the free energy of the cross-bridge in state i (U_i) is a linear combination of the strain-dependent and phosphate-dependent energy of the cross-bridge. The free energies of the 4sXB system are:

$$\begin{aligned} U_1(\alpha,\beta,\delta,\gamma) &= 0 \\ U_2(\alpha,\beta,\delta,\gamma) &= \alpha_e \Delta G + \frac{k_\alpha(\alpha-\alpha_0)^2 + k_\beta(\beta-\beta_0)^2 + k_\delta(\delta-\delta_0)^2 + k_\gamma(\gamma-\gamma_0)^2}{2} \\ U_3(\alpha,\beta,\delta,\gamma) &= \eta_e \Delta G + \frac{k_\alpha(\alpha-\alpha_0)^2 + k_\beta(\beta-\beta_0)^2 + k_\delta(\delta-\delta_1)^2 + k_\gamma(\gamma-\gamma_0)^2}{2} \end{aligned} \tag{2.1}$$

The free energies of the 2sXB system are:

$$\begin{aligned} U_1(\theta,\rho) &= 0 \\ U_2(\theta,\rho) &= \alpha_e \Delta G + \frac{k_\rho(\rho-\rho_0)^2 + k_\theta(\theta-\theta_0)^2}{2} \\ U_3(\theta,\rho) &= \eta_e \Delta G + \frac{k_\rho(\rho-\rho_1)^2 + k_\theta(\theta-\theta_1)^2}{2} \end{aligned} \tag{2.2}$$

Binding rate calculation Our binding algorithm follows Tanner et al. (2007) [Tanner et al., 2007] but differs in two key ways: (1) we split binding to the thin filament into two steps, and (2) our diffusion step works with any number of springs. Previous models treated binding rate constants as an exponential function of the distance between a tethered

diffusing spring and an available binding site [Tanner et al., 2007, Daniel et al., 1998]. We produce binding rate constants in the same fashion, but in adapting them for multi-spring cross-bridges, split the process into two steps: diffusion of the myosin head to a new location, followed by calculating binding probability at the new location. The energy of a single spring undergoing thermally forced diffusion is taken from a Boltzmann distribution of possible energies [Berg, 1993, Howard, 2001]. With a single-spring cross-bridge, the cross-bridge tip offset is easily found from the energy of the spring. A thermally forced multi-spring cross-bridge likewise takes the energy for each of its constituent springs from a Boltzmann distribution of energies, but tip location must be separately calculated (see the Geometry section above). It is this separation of the calculation of cross-bridge tip location from binding probability that splits our binding rate calculation into two steps.

In the diffusion step, each spring is offset from rest with an energy taken from the probability density function: $P(x) = \sqrt{k/(2\pi\kappa_b T)}\exp^{-(kx^2)/(2\kappa_b T)}$ where x is the offset, k is the spring constant of the particular spring, κ_b is the Boltzmann constant, and T is the system's temperature in Kelvin [Dill and Bromberg, 2003, Howard, 2001]. The new spring values are used to update the location of the cross-bridge tip, which is used to calculate the post-diffusion distance, d_{diff}, from the tip to the binding site of interest. As in previous models, the probability that the cross-bridge will bind to a given binding site decreases exponentially as d_{diff} increases. Thus the probability a cross-bridge will bind to an available site is given by $p_{12}(d_{\text{diff}}) = \tau \exp^{-d_{\text{diff}}^2}$, where τ is a scale factor with a value of 12 for the 4sXB and 72 for the 2sXB, chosen to provide attachment rates consistent between the multiple spring cross-bridge models. Attachment occurs when p_{12} is greater, on a given 1 ms time step, than a random number chosen from a uniform distribution in the domain 0 to 1 [Tanner et al., 2007]. This process is sufficient to determine if a cross-bridge in simulation binds in a given time step, but binding rate constants, as used in Figures 2.2 and 2.3 are calculated with an ensemble of cross-bridges. Thus, for an ensemble of size n:

$$r_{12} = \frac{\sum_0^n \left(1 \text{ if } \tau \exp^{-d_{\text{diff}}^2} > rand, \text{ else } 0\right)}{n} \tag{2.3}$$

This two step system, with diffusion followed by a chance of attachment, is used for both

the 4sXB and 2sXB models with only a change in the number of thermally forced springs and the scaling factor τ.

Power stroke and detachment rates The power stroke and detachment rates are adaptations of prior models [Pate and Cooke, 1989, Tanner et al., 2007]. Unlike with binding, the power stroke and detachment rate constants explicitly depend on the free energy of the cross-bridge. This free energy is calculated from equations 2.1 and 2.2, with the tip co-located at the relevant binding site. In the case of the 4sXB, each calculation of a transition rate constant requires that the location of the converter domain be optimized to relax the cross-bridge into its lowest energy state. Of note, the unbound cross-bridge supports no strain and so $U_1 = 0$. These rate constants are insensitive to the number of springs comprising each cross-bridge and function similarly in one- and two-dimensional models. Both the power stroke rate constant (r_{23}) and the detachment rate constant (r_{31}) depend on the differences in free energy between the current state and the one being considered for transition. This dependence on the difference in free energies means transitions are more likely when they are energetically favorable and less likely in other circumstances, a natural scheme based in the geometry of the cross-bridges. The particular rate constants for both the 4sXB and the 2sXB models are:

$$r_{23}(U_2, U_3) = 100 + 100 \tanh(4 + 0.4(U_2 - U_3)) \tag{2.4}$$

$$r_{31}(U_3, U_1) = 20 + 100(U_3 - U_1)^{1/2} \tag{2.5}$$

Calculation of reverse rates The reverse transition rate constant from state i to state j is given by the thermodynamically balancing formula:

$$r_{ij}/r_{ji} = \exp^{U_i - U_j} \tag{2.6}$$

where r_{ji} is the forward rate constant and r_{ij} is the reverse rate constant [Pate and Cooke, 1989, Daniel et al., 1998, Tanner et al., 2007]. The transition from a pre-power stroke state to an unbound state requires the reverse transition again be treated as a fraction of an

ensemble of transition opportunities, using Equation 2.3 to provide r_{12}. The remaining forward transition rate constants, r_{23} and r_{31}, are calculated from equations 2.4 and 2.5, while all free energies are provided by equations 2.1 and 2.2.

2.7 Acknowledgments

Support was provided by an NIH pre-doctoral training grant to CDW and by the Joan and Richard Komen Endowed Chair to TLD

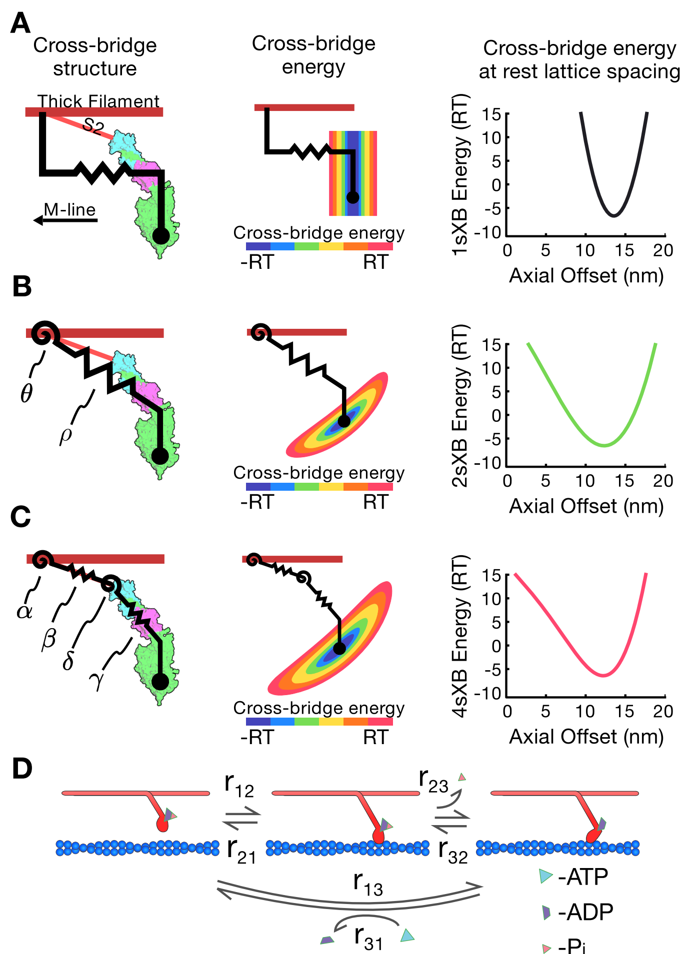


Figure 2.1: **Cross-bridge types and kinetic scheme.** **A** through **C** show the three cross-bridge models, plotted against a myosin crystal structure for comparison (structure image generated from Gourinath et al. (2003) [Gourinath et al., 2003] with PyMol [Delano, 2008]). The energy landscape of each cross-bridge and the free energy at rest lattice spacing are shown adjacent to the cross-bridge schematic. **A)** The 1sXB introduced in Huxley (1957) [Huxley, 1957]. **B)** The 2sXB which uses a torsional/angular spring (θ) and an extensional spring (ρ). **C)** The 4sXB with two torsional and two extensional springs. Of the 4sXB's springs, α corresponds to the point at which the S2 region rejoins the thick filament backbone, β to the S2 region itself, δ to the area linking the S2 and the light chain domains, and γ to the light chain domain itself. δ replicates the change in angle accompanying the power stroke by applying torque to the freely moving joint representing the converter domain. **D)** The three state kinetic system. The three states represent (1) an unbound state, (2) a pre-power stroke state, and (3) a post-power stroke state. The rate of transition between states i and j is represented as r_{ij}. The forward and reverse transition rate constants are functions of energy stored in the cross-bridge.

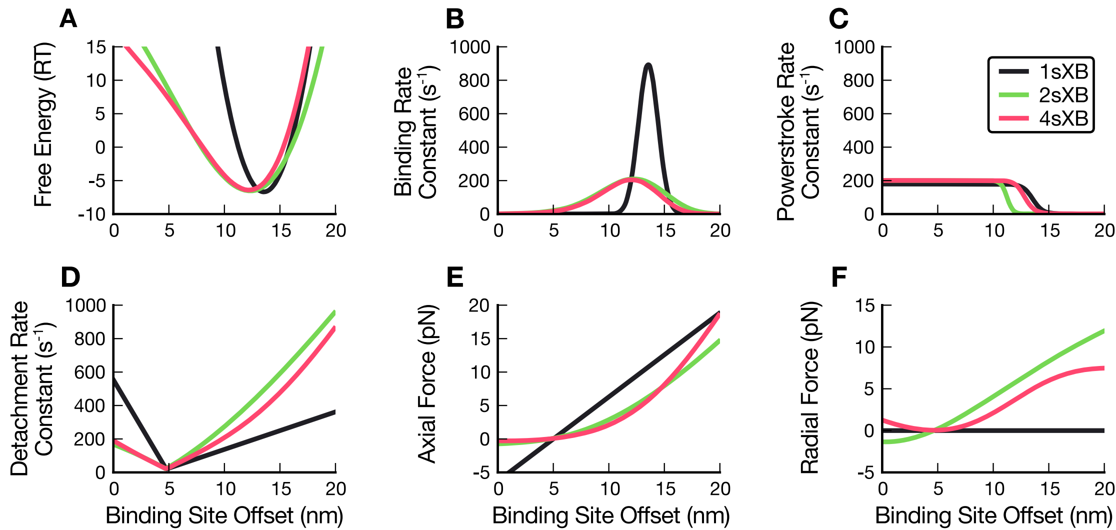


Figure 2.2: **Forces, energy, and kinetics of the 1sXB, 2sXB, and 4sXB models at resting lattice spacing.** The energy, transition rate constants, and forces of the 1sXB model (black), 2sXB model (green), and 4sXB model (red) at shown at resting lattice spacing. The 1sXB model values shown for comparison are derived from those of Daniel et al. (1998) and Tanner et al. (2007), [Daniel et al., 1998, Tanner et al., 2007], shifted axially so the resting location of the cross-bridge head in each case is aligned with the resting locations of the 2sXB model and 4sXB model allowing easier comparison. The free energy of the cross-bridges in state two is shown in **A**, where the multi-spring cross-bridges' shifts from a purely parabolic trajectory is visible. The explicit two-dimensional thermal forcing of the multi-spring cross-bridge heads in **B** results in binding probabilities that are more distributed than those of the single spring cross-bridge. The rate of power strokes **C** remains least changed between the single and the multi-spring cross-bridge models. The energy-based kinetics of the multi-spring cross-bridges are unable to fully replicate the biased detachment rate of the 1sXB model in **D**. **E** and **F** show the 1sXB's sharp discontinuities in axial force and lack of any radial force.

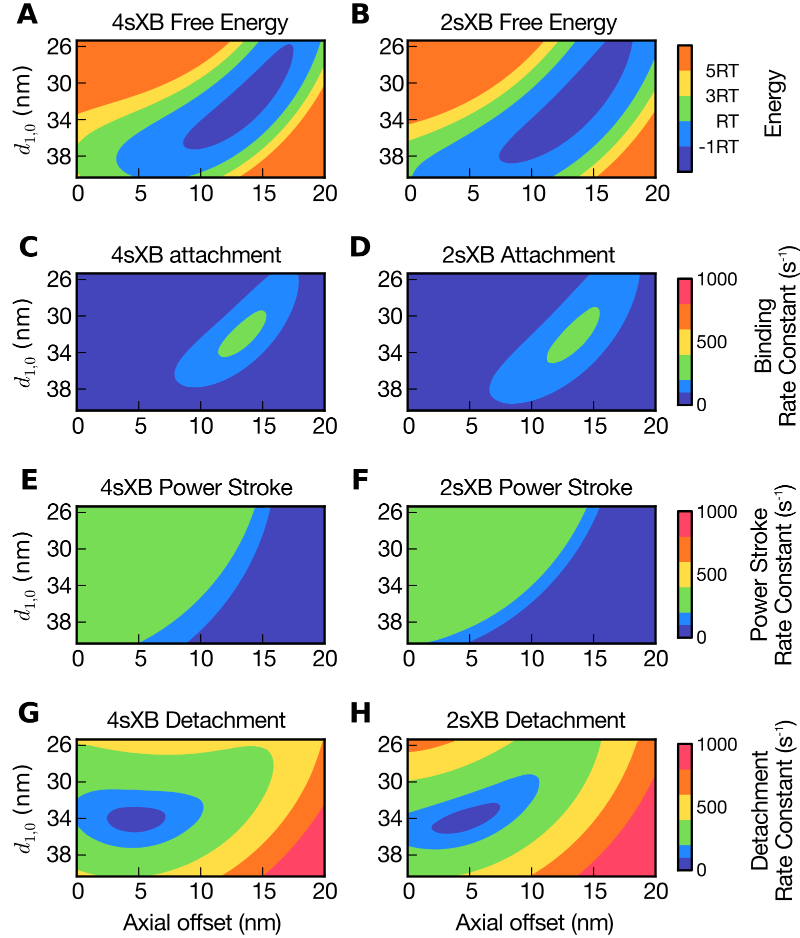


Figure 2.3: **Energy and kinetics of the multi-spring cross-bridge models change with axial offset and lattice spacing.** Axial offset is the distance between the current axial location of the cross-bridge tip and the location where the cross-bridge attaches to the thick filament. Lattice spacing (d_{10}) is defined as in Millman (1998) [Millman, 1998], with an offset to account for filament thicknesses so the cross-bridge spans the filaments at a rest lattice spacing of 34 nm. Panels depict the properties of the 4sXB model (**A**, **C**, **E**, and **G**) and the 2sXB model (**B**, **D**, **F**, and **H**) as they change with binding site offset and lattice spacing. **A** depicts the free energy of the 4sXB model at various lattice spacings, with the head stretched to an axial offset from the thick filament attachment point. The free energy of the 2sXB model is shown in **B**. **C** and **D** show r_{12}, the probability that the 4sXB and 2sXB models will transition from an unbound state to a bound state. **E** and **F** show r_{23}, the probability of transition from a pre-power stroke state to a post-power stroke state, for the same cross-bridges, axes, and scales as **C** and **D** show r_{12}. **G** and **H** show r_{31}, the probability of unbinding from a post-power stroke state. The reverse rate constants, r_{21}, r_{32}, and r_{13} are back-calculated from the forward rate constants.

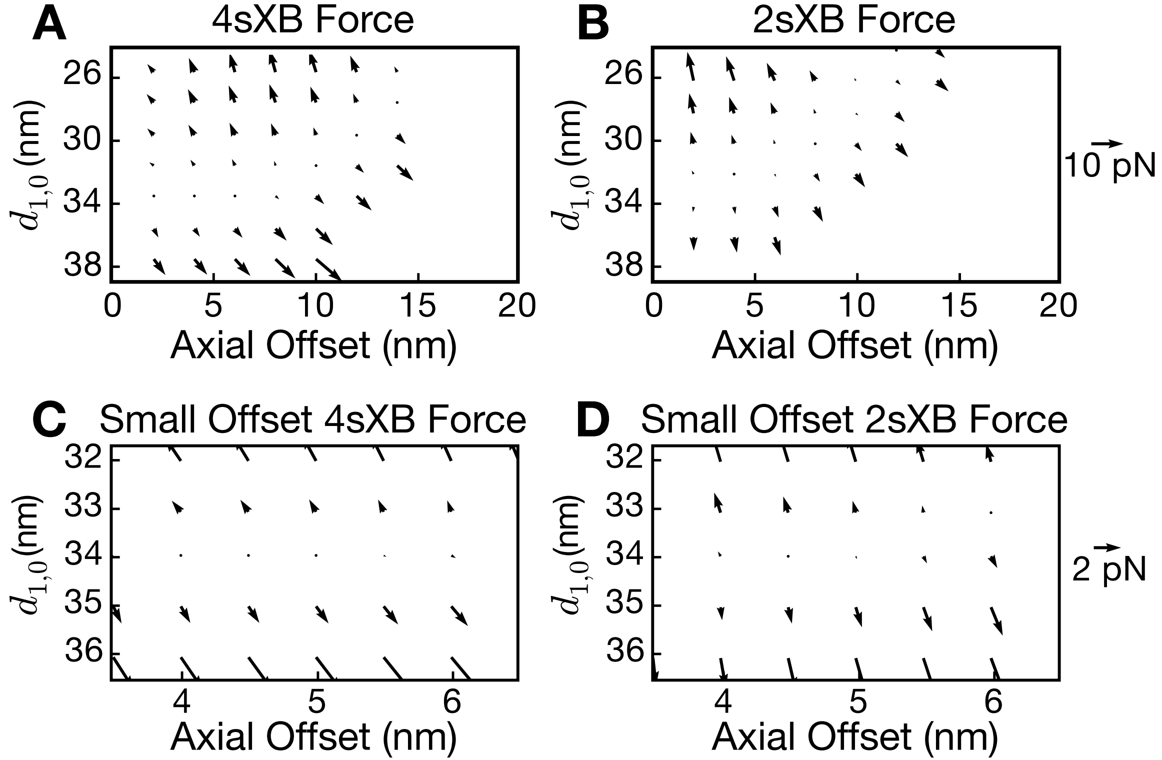


Figure 2.4: **Overview and detail of the forces exerted by the 2sXB and 4sXB models in the post-power stroke state.** The post-power stroke forces exerted by the 4sXB and the 2sXB models are shown as vector fields of reaction forces. The reaction force is that necessary to retain the cross-bridge head in a given location, thus the vectors for a compressed cross-bridge orient upwards and those for an extended cross-bridge orient downwards. Positions in which the cross-bridge is unlikely to generate force are omitted; these unlikely locations are determined by the sum of r_{23} and the inverse of r_{31}. **A** and **B** show overviews of the forces exerted, respectively, by the 4sXB model and the 2sXB model over lattice spacings and axial offsets that vary as in Figure 2.2. The forces exerted by the two cross-bridges have radial components which frequently equal or exceed their axial components. A more detailed view of the region surrounding the rest position of the cross-bridges is shown in **C** and **D**, where the large radial components of the cross-bridge forces, particularly for the 2sXB model, is especially evident.

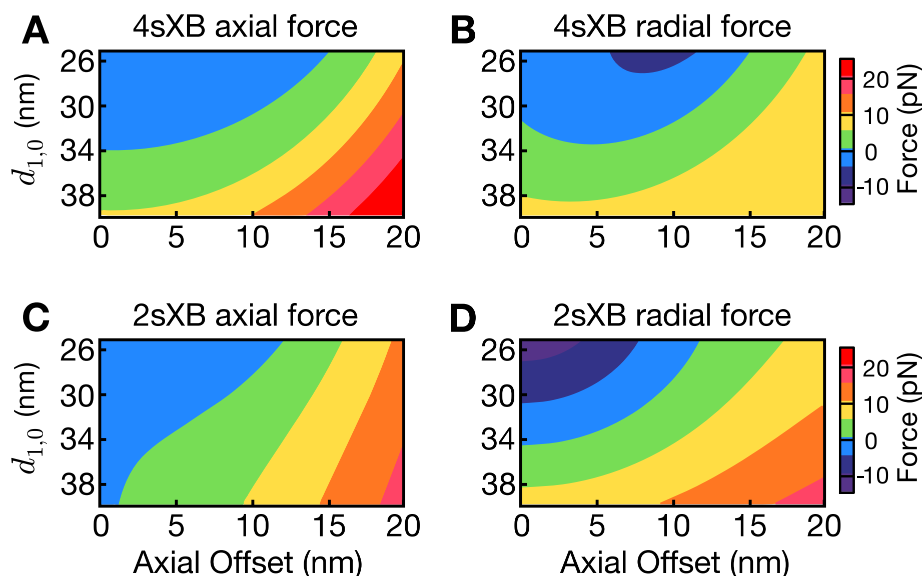


Figure 2.5: **Axial and radial post-power stroke forces as separate components. A** through **D** show, separated, the axial and radial components of the forces produced by the 4sXB and the 2sXB models in the post-power stroke state.

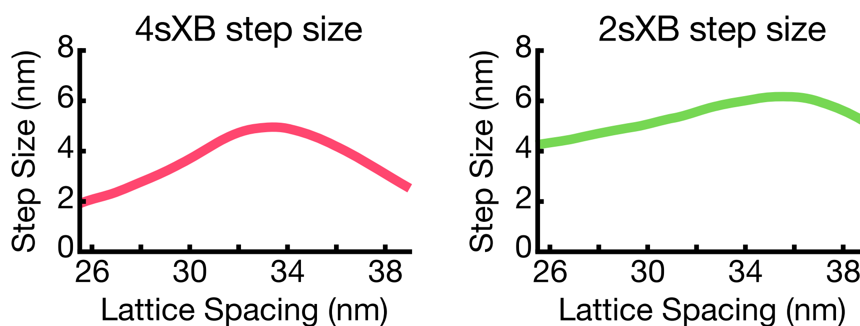


Figure 2.6: **Changes in step size with lattice spacing.** Step size varies as lattice spacing diverges from its rest value. Step size is defined as the change in the rest axial offset between the pre- and post-power stroke states. The step size of the 4sXB model and 2sXB model produce different absolute step sizes as lattice spacing change. However, both models exhibit a local maximum step size at a specific lattice spacing with a decreasing step size as lattice spacing diverges from that point.

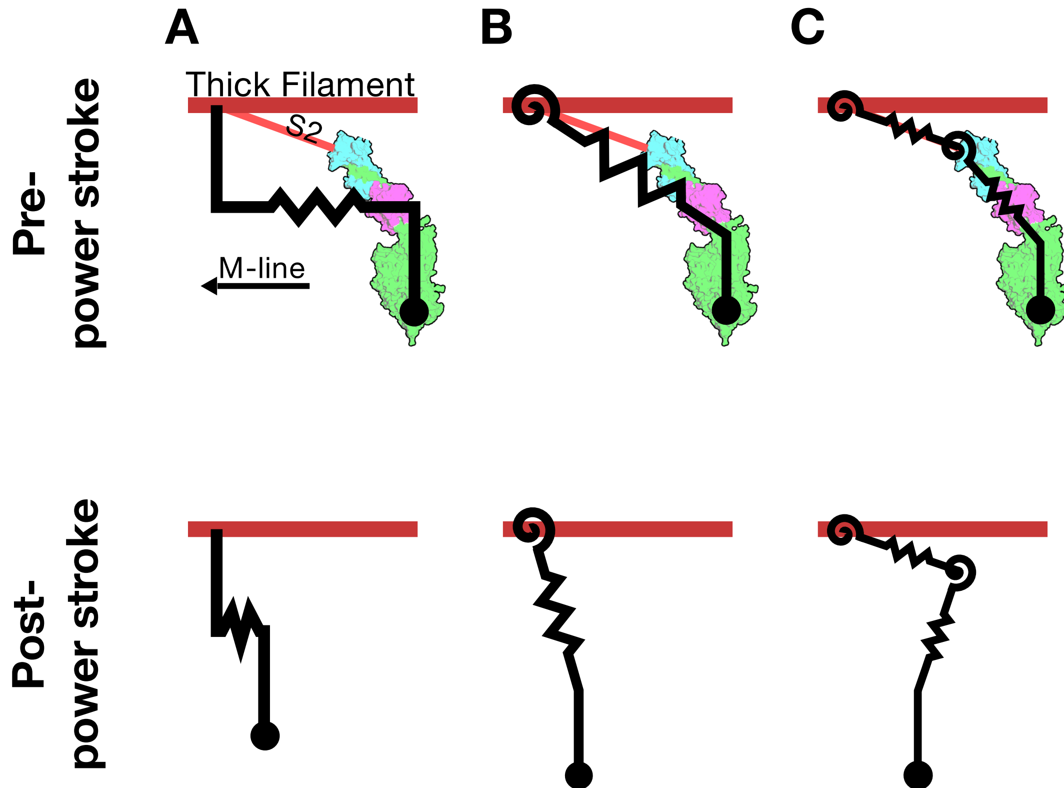


Figure 2.7: **Changes in cross-bridge resting geometry with the power stroke. A** through **C** show, schematically, the change in the rest lengths and angles of the single and multi-spring cross-bridges. The rest length and angle of the 2sXB's extensional and torsional springs are set, in both the pre- and post-power stroke states so as to match the tip position of the 2sXB in each condition to that of the 4sXB (in Table 2.1). The change in the unstressed radial distance from the thick filament to the tip of the multi-spring cross-bridges that occurs with the power stroke is particularly visible in **B** and **C** when compared to the single spring cross-bridge **A**. The effects of the universal joint attaching the springs of the 4sXB and the 2sXB to the globular domain, and the globular domain's own fixed angle to the thin filament, are shown by the continued radial orientation of the globular domain after the power stroke occurs.

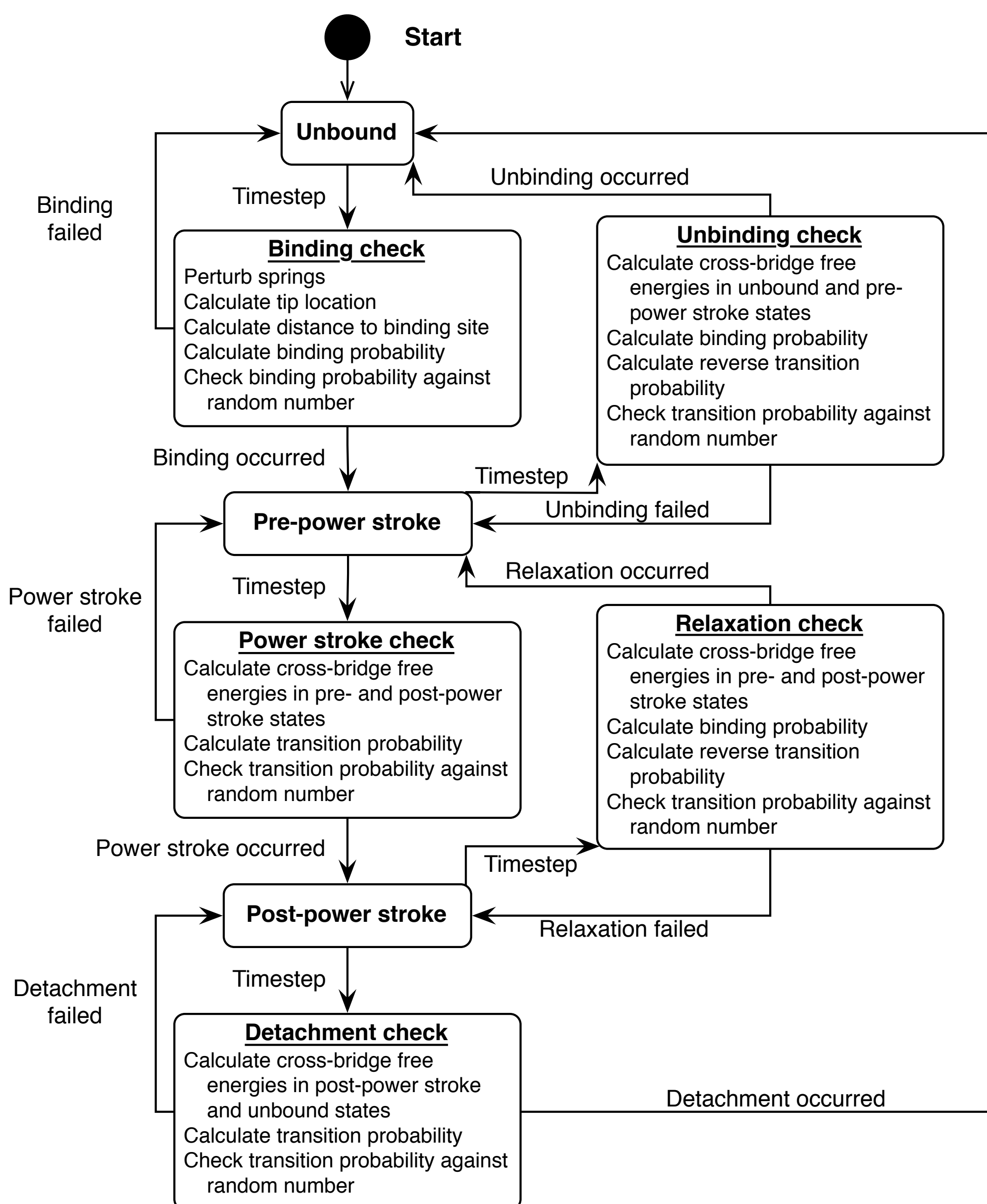


Figure 2.8: **Model simulation protocol** The model simulation process, as described throughout the paper, is displayed as a state diagram. Entering the diagram at "Start", the states and actions which change those states are depicted for a single cross-bridge.

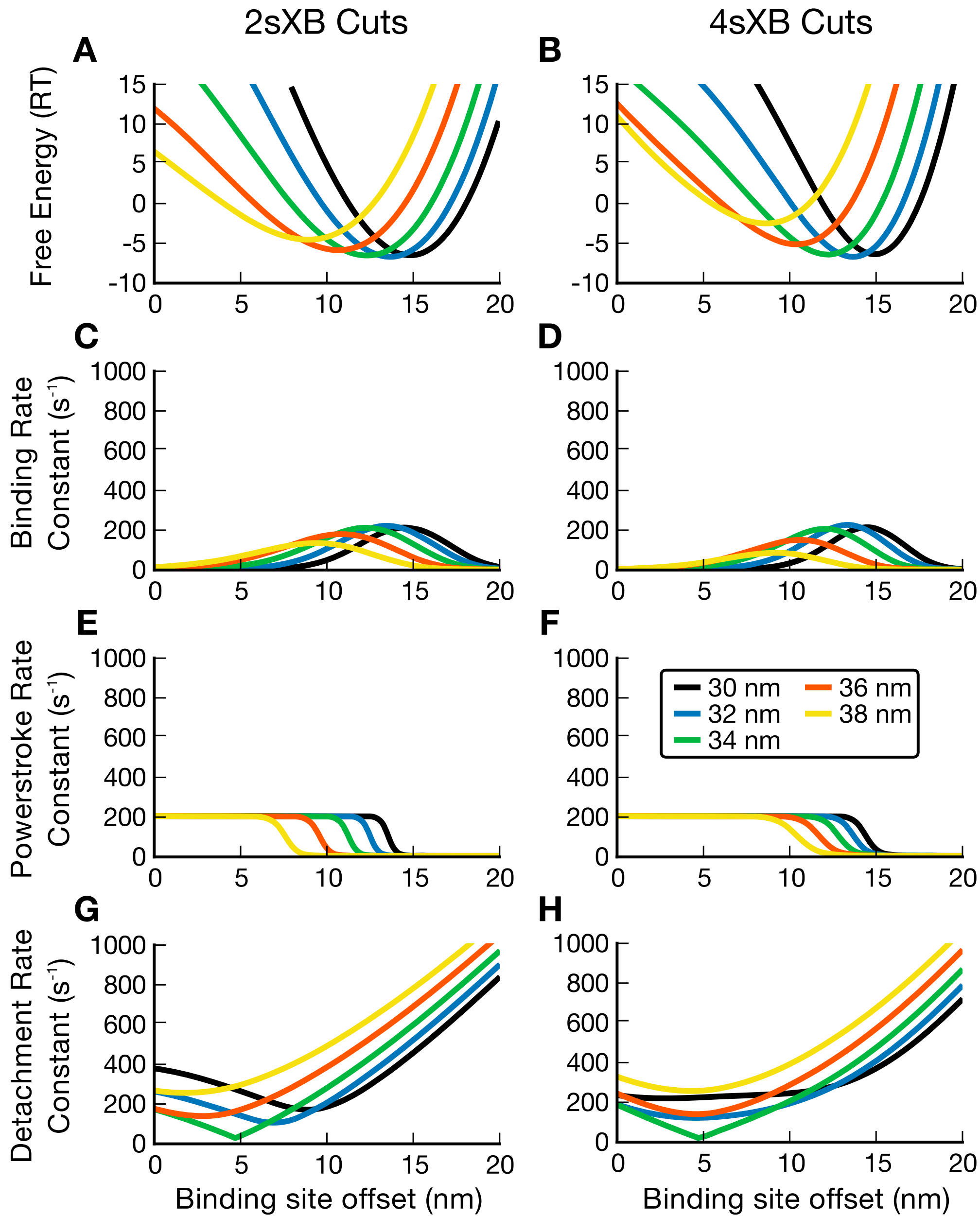


Figure 2.9: **Cross-bridge free energy and kinetics at multiple lattice spacings A** and **B** show the free energies of the 4sXB and the 2sXB models at lattice spacings between 30 and 38 nm. **C** through **H** show the kinetic rate constants of the 4sXB and the 2sXB models at lattice spacings between 30 and 38 nm. Each rate is a section taken from the corresponding display in Figure 2.3.

Table 2.1: **Model parameters and their sources**

Model	Spring	Rest value	k	Source
4sXB	α	40°	100 pN/rad	Liu et al. [2006]
	β	10.5 nm	10 pN/nm	Liu et al. [2006]
	δ	125°	40 pN/rad	Taylor et al. [1999]
	δ'	70°	40 pN/rad	Taylor et al. [1999]
	γ	9.6 nm	5 pN/nm	Houdusse et al. [2000]
2sXB	θ	47°	40 pN/rad	See caption
	θ'	73°	40 pN/rad	See caption
	ρ	20 nm	2 pN/nm	See caption
	ρ'	16 nm	2 pN/nm	See caption
1sXB	k	5 nm	5 pN/rad	Tanner et al. [2007]
	k'	0 nm	5 pN/rad	Tanner et al. [2007]

Prime values, such as δ', represent post-power stroke state values. From Liu et al. (2006) [Liu et al., 2006], which used insect flight muscle, the most frequently occurring thick filament to S2 angle range is 51-60°. We assume that this range is being distorted by the compressive radial force being generated by the rigor cross-bridges in the swollen lattice spacings that Liu et al. used. As such, we choose a rest angle for α at the low end of the still common range of 50° to 40°. We do not change this angle between states one, two and three. In Taylor et al. (1999) [Taylor et al., 1999] (clearly explained in [Davis and Epstein, 2009]) the angle between the LCD and the thick filament's axial axis goes from 125° to 70° with the power stroke. The LCD rest length generated by measurements made of structure 1DFK from Houdusse et al. (2000) [Houdusse et al., 2000]. The rest values of the 2sXB model's springs are determined by those of the 4sXB model; they are calculated so that the rest position of the 2sXB's head is the same as the rest position of the 4sXB's head. The spring constant, k, for the angular spring responsible for each cross-bridge's power stroke is determined by the change in angle over the power stroke and the energy liberated by the hydrolysis of ATP [Tanner et al., 2007]. Additional spring constants are chosen to be consistent with previous work, and to provide sufficient flexibility to enable diffusion. The parameters of the single spring cross-bridge, used for comparison, are taken from [Tanner et al., 2007].

Chapter 3

THE SLOPE OF THE LENGTH-TENSION CURVE IN MUSCLE DEPENDS ON LATTICE SPACING.

C David Williams, Mary K Salcedo, Thomas C Irving, Michael Regnier, Thomas L Daniel

3.1 Abstract

Classic interpretations of the striated muscle length-tension curve focus on the role that changing thin (actin) and thick (myosin) filament overlap plays in generating force. New models of sarcomere geometry and experiments with skinned insect flight muscle suggest that changes in the radial distance between the actin and myosin filaments, the filament lattice spacing, are responsible for between 20% and 50% of the change in force seen between sarcomere lengths of 1.4 μm and 3.4 μm. This places lattice spacing in the role of a significant force regulator, increasing the slope of the force-length dependence of muscle.

3.2 Introduction

That muscle maintains a constant volume during contraction had been known since the mid-1600s [Cobb, 2002]. However, since the 1950s, skinned muscle fibers have provided much of our insight into the operation of the molecular machinery which generates force in muscle. These skinned fibers have had their surrounding membranes removed or disrupted and so, unlike intact muscle, don't necessarily maintain a constant volume.

The length-tension (LT) curve has long been the basis for how we interpret the basic mechanism of force generation by actin and myosin [Gordon et al., 1966]. Experimental data and theoretical work are described in terms of events occurring in the axial direction, aligned with muscle's thick and thin filaments [Rassier et al., 1999]. We have long known that striated muscle is able to generate different levels of force at different lengths [Blix, 1894]. The combination of this longstanding knowledge with the sliding-filament theory has pushed forward our understanding of muscle contraction [Huxley and Hanson, 1954, Huxley

and Niedergerke, 1954, Gordon et al., 2000]. However, this body of classic work has not considered the effects of changes in lattice spacing, the radial distance between thick and thin filaments, caused by changes in muscle length [Millman, 1998, Irving et al., 2000]. Here we show lattice spacing's unrecognized and critical role in determining the length-tension curve of contracting muscle.

The LT curve has three major sections: the ascending limb, the plateau region, and the descending limb. The relationship between force and sarcomere length has a distinct slope in each of these sections. Changes in sarcomere length alter the degree of overlap of the thick and thin filaments, which is thought to be the regulator of how many cross-bridges can generate force. The number of cross-bridges that can bind and generate force determines the total force produced by the muscle.

When a muscle fiber's sarcomeres are stretched to long lengths, the fiber is on the descending limb of the LT curve. At these lengths there is little overlap of the thick and thin filaments. This small degree of overlap places few myosin heads opposite a thin filament; thus few cross-bridges can form and generate force. As muscle shortens, sarcomere length decreases and more myosin heads are located opposite a thin filament. This allows more cross-bridges to form and generate force. When the maximal number of myosin heads have access to an opposing thin filament, the fiber has entered the region of maximal force production: the plateau region of the LT curve. When the fiber shortens beyond the plateau region, thin filaments from opposite Z-disks increasingly overlap and sarcomere force declines. This is the beginning of the ascending limb of the LT curve. It is generally thought that this overlap of thin filaments shields binding sites from myosin heads, resulting in decreased formation of cross-bridges and decreased force.

Explanations of the LT curve based purely on processes in the axial direction are appealing, in part because the known lengths and dimensions of the thick and thin filaments can predict the sarcomere lengths where the LT curve transitions between phases [Rassier et al., 1999]. But as sarcomeres shorten, in addition to an increase in overlap of the thick and thin filaments, the radial distance between filaments also increases. The radial distance between the faces of neighboring thick and thin filaments, here referred to as lattice spacing, affects the kinetics of attachment and detachment of the myosin heads [Adhikari

et al., 2004, Fuchs and Martyn, 2005, Williams et al., 2010]. These altered kinetics are often invoked to explain, e.g., changes in Ca^{2+} sensitivity [Godt and Maughan, 1981, Fuchs and Wang, 1996]. An increase in lattice spacing implies that there is a greater distance between the unstrained location of a myosin head and a potential binding site, the distance which a head must diffuse to bind. Additionally, when cross-bridges do form, the angle of the S2 domain which mechanically couples the head to the thick filament varies with lattice spacing [Schoenberg, 1980a, Williams et al., 2010]. The S2 angle alters both the force that can be generated by the cross-bridge and this force's vector at the thick filament. Thus, as muscle moves along the LT curve and diffusion distance and attachment angle change, it is reasonable to assume their variation plays a role in determining cross-bridge kinetics and force production. This leads us to ask, what fraction of the force changes on the ascending and descending limbs can be explained by changes in lattice spacing?

Evidence for a change in lattice spacing with a change in sarcomere length comes from X-ray diffraction measurements and the need for muscle to obey (if only approximately) the isovolumetric condition [Millman, 1998, Irving et al., 2000]. The isovolumetric condition posits that the lattice volume of muscle is constant over the time scale of contractions, implying there is no bulk flow of fluids into or out of the myofibrils. Aside from the difficulty in removing fluid from a tightly packed lattice, the sarcolemma surrounding each fiber enforces a constant cell volume at the sub-second timescale. While the fiber maintains a constant volume over the timescale of a contraction, this constraint is slightly relaxed over longer durations [Rapp et al., 1998].

Here we address the ways in which lattice spacing regulates maximum force production across the LT curve, through a combination of skinned fiber X-ray diffraction experiments and a multiple filament, spatially explicit model of the sarcomere that incorporates lattice spacing (Figure 3.1A). We compare our spatially explicit model against both new data and classic data from literature [Gordon et al., 1966]. We demonstrate that changes in lattice spacing substantially increase the length-dependence of force. Thus the slope and shape of the LT curve is a product of both the axial and radial geometry of contracting muscles (Figure 3.1B).

3.3 Results

Changing lattice spacing increases the slope of the LT curve (Figures 3.2 &3.3). Changing only sarcomere overlap produces a shallow force-length relationship, reproducing 75%-81% of the force change seen under isovolumetric conditions. Osmotically changing only lattice spacing in skinned fibers reproduced much of the force change expected from isovolumetric conditions (Figure 3.4).

3.3.1 Theoretical and experimental isovolumetric LT curves agree.

The force produced by our model under isovolumetric conditions closely matches published actively-generated force levels for frog striated muscle (Figure 3.2) [Gordon et al., 1966]. As shown in Figure 3.2A&D, the model produces 18-21% of its peak force for simulations at very short or very long sarcomere lengths of 1.4μm or 3.4 μm, while isolated striated muscle fibers generate between 25% and 30% of their peak forces when contracted at the same lengths. These comparisons address only actively generated forces, as our model does not include passive force generating elements such as titin.

Transitions between different phases of the LT curve occur at similar sarcomere lengths in both isolated fibers and our model. In both experimental data and model results, the steepest part of the ascending limb transitions into the start of the plateau region at sarcomere lengths 1.8$\pm$0.1μm and the descending limb begins at 2.3$\pm$0.1μm. In neither instance does the model disagree with the experimental results by more than the uncertainty inherent in estimating the transitions between phases of the LT curve. Agreement between our computational results and the force generated by isolated fibers gives us confidence to use the force generated by our model under isovolumetric conditions as a basis to which we compare constant length and constant lattice-spacing forces. These comparisons are shown for the LT curve in Figure 3.2 and the relative force regulation of both changes in lattice spacing and degree of overlap is shown for all parameters in Figure 3.3.

3.3.2 Changing only lattice spacing alters force similar to the isovolumetric LT curve.

Changing only lattice spacing, while holding filament overlap constant, is sufficient to recapture much of the force change along the isovolumetric LT curve (Figure 3.2A&B). To isolate the effects of lattice spacing on isometric maximum force, the model allows us to hold sarcomere length at a constant value, 2.3 μm in this case, while changing lattice spacing to the values present in an isovolumetric LT curve. This allowed us to compare the effects of an isolated change in lattice spacing to those of an equal change in lattice spacing produced by isovolumetric conditions.

On the ascending limb, as the distance over which myosin heads must diffuse in order to bind and generate force grows, the force generated decreases. While the overlap and thus the number of myosin heads with an opposing thin filament remains constant, the force generated drops to 47.1% of its peak value at a lattice spacing of 21.9 nm. Combined shielding of binding sites with increased lattice spacing present in isovolumetric conditions produce a drop to 17.6% of peak force; changing only lattice spacing can cause over 60% of the force change seen in the isovolumetric ascending limb.

On the descending limb the lattice spacing decreases below its plateau-region value and cross-bridges become less likely to form and generate force. The force at 14.1 nm, the smallest lattice spacing (see Figure 3.7B for definitions), dips to 72.0% of the maximum force. With the equivalent isovolumetric point producing just 21.4% of the maximal force, lattice spacing change reproduces 36% of the isovolumetric force drop and is thus clearly less influential on the descending limb.

3.3.3 Changing only overlap reproduces more of the isovolumetric force change.

When lattice spacing is held constant and only sarcomere length is changed, the classic LT curve is followed, but is less steep than under isovolumetric conditions (Figure 3.2A&C). In this case, changes in sarcomere length caused filament overlap and binding site availability to vary, but the lattice spacing was fixed at 17.1 nm.

On the ascending limb, below sarcomere lengths of 1.8 μm, changing only length drops the force produced to 37.3% of the maximum value. This decrease is 74% of what occurs in

the isovolumetric case, slightly greater than the 60% reproduction of the isovolumetric case created by altering only lattice spacing.

On the descending limb, as sarcomere length increases beyond 2.3 μm, the overlap of the thin filament and the myosin S1 decorated regions of the thick filament lessen. As sarcomere length approaches 3.5μm, the force drops to 35.2% of its peak; 81% of the drop under isovolumetric conditions. The difference between the force drop which occurs under isovolumetric conditions and the force drop when only lattice spacing is changed is more than twice as big as the difference between the force drop under isovolumetric conditions and the force drop when only overlap is changed.

The force-length relationship seen under simulated and observed isovolumetric conditions is better replicated by changing only sarcomere length than it is by changing only lattice spacing. Changing only overlap better predicts the force decrease at extreme lengths. In addition the transitions between the different phases of the LT curve are more sharply defined in the case of overlap change than they are in the case of lattice spacing change, although isolating either property still does a worse job of predicting the observed LT curve than when both lattice spacing and sarcomere length change isovolumetrically.

3.3.4 Lattice spacing tunes force at all overlaps and vice versa.

Our results show that lattice spacing partially regulates the level of force produced at every sarcomere length and, conversely, sarcomere length partially regulates the level of force produced at every lattice spacing. This mutual dependence can be seen in the shape of Figure 3.3, which is peaked rather than a section of a cylinder. The effect this has on force is seen by tracing the paths of the three cases examined in Figure 3.2 across the force surface of Figure 3.3. An isovolumetric LT relationship is simultaneously descending two slopes, the slope of lattice spacing and the slope of sarcomere length. Thus it is steeper than if it was descending only one slope. The reduced slope of the descending limb relative to that of the ascending limb is also explained by the square root dependence of lattice spacing on length. As sarcomeres lengthen on the descending limb, lattice spacing diverges more slowly from its optimum value than when sarcomeres shorten on the ascending limb.

3.3.5 Decreasing lattice spacing reduces force on skinned fibers' descending limb.

Skinned bundles of *Manduca sexta* flight muscle fibers stretched to a sarcomere length of 4.0±0.2 μm reproduce the LT relationship of the descending limb purely through osmotic compression of the filament lattice (Figure 3.4B). Force decreases to 20% of its maximum value with a reduction in lattice spacing equivalent to that seen as vertebrate muscle lengthens from 2.3 μm to 3.4 μm. This is greater than the predicted constant-length force decrease in Figure 3.2B. The variance in initial sarcomere lengths between 3.8 μm and 4.2μm is not a significant factor, maximum force is independent of influence from this noise (Hoeffding's D value -0.015, p=0.72).

3.4 Discussion

3.4.1 Lattice spacing amplifies length-based force regulation.

This work demonstrates that lattice spacing has a substantial influence on the length-tension (LT) curve. Lattice spacing change over the course of the LT curve is not the primary determinant of the force output, overlap has a far greater effect on the force produced, but changing lattice spacing augments the effects of changes in length. Changing lattice spacing increases the steepness of the LT curve by altering myosin kinetics and the force generated by attached cross-bridges [Williams et al., 2010]. The LT curve would be 20% shallower without the effects of lattice spacing, and the regulation potential of length changes would be commensurately reduced. Effects such as the Frank-Starling mechanism in cardiac muscle rely on a steep slope on the ascending limb to passively regulate the amount of force produced [Smith et al., 2009]. Without the effects of lattice spacing or some additional form of regulation, processes such as the Frank-Starling mechanism would not remain in a stable equilibrium.

3.4.2 Lattice spacing regulation extends into the descending limb.

It is interesting that the effects of lattice spacing extend into the descending limb where, unlike with the large lattice spacings in the ascending limb, myosin heads are not required to bridge a large distance in order to bind and generate force. This is likely due to the

geometric restrictions such small lattice spacings place on the cross-bridge (Figure 3.1B). The limited radial distance over which a myosin diffuses on the descending limb shortens the section of the opposing thin filament it can access. These effects do not reduce force as quickly as does the change in lattice spacing along the ascending limb in part because of the square root dependence of lattice spacing on sarcomere length. Lattice spacing changes less with a given length change at longer lengths than at shorter lengths [Millman, 1998, Irving, 2006]. The observed lattice spacing dependence of skinned M. sexta muscle over the descending limb suggests the lattice spacing-force relationship observed in our simulations, if anything, somewhat underestimates the degree to which lattice spacing affects maximum force.

3.4.3 Simple geometry produces complex regulation.

The geometry of the thick-thin filament interaction is a regulator of force. Changes in the radial direction, such as those due to altered lattice spacing, have effects of the same order of magnitude as those in the axial direction. Including these relatively simple geometries in our mathematical and conceptual models of sarcomeric force generation demonstrates an important geometric regulation of force regulation.

The fact that lattice spacing changes over the LT curve has been known for some time, but its importance in generating the steep length dependence of force has not been previously demonstrated. Given the importance of the LT curve in both functional processes and as a diagnostic of the cross-bridge cycle, this work forms a new basis of interpretation.

3.5 Methods

3.5.1 Single myosin model

Our theoretical results are produced using a conceptually new model of the half-sarcomere (Figure 3.5A). This half-sarcomere is populated with two-spring models of the cross-bridge as described previously [Williams et al., 2010]. The two-spring cross-bridge generates force through a change in angle matched to myosin's lever arm mechanism. These cross-bridges are sensitive to changes in lattice spacing because they are modeled in two dimensions,

rather than because of any explicit lattice-spacing-dependent terms. As in our prior work, lattice spacing is measured from the face of a thick filament to that of an opposing thin filament (Figure 3.7B) [Williams et al., 2010]. At resting lattice spacing, these cross-bridges' kinetics are derived from, and are compatible with the kinetics of prior models [Tanner et al., 2007, Pate and Cooke, 1989]. The force produced by these cross-bridges also depends on the lattice spacing of the system as it affects the angle at which they attach, and thus the deviation of their constituent springs from their rest values.

3.5.2 Multifilament model's geometry

A semi-infinite muscle lattice is created with toroidal boundary conditions as in prior work [Tanner et al., 2007]. As shown in Figures 3.1A&3.5 B, when a cross-bridge passes through a boundary it wraps around to the other side of the model. A total of four thick and eight thin filaments suffice to create a semi-infinite lattice where no thick filament is attaching to a neighboring thin filament from two directions (once directly and once across a boundary) and where the ratio of thick to thin filaments is the same as in skeletal muscle (Figure 3.1A).

Each thick filament is populated by a 43 nm repeating pattern of cross-bridges [Tanner et al., 2007]. Each repeating unit has three evenly spaced axial crowns of three cross-bridges each [Tanner et al., 2007]. The cross-bridges in a crown are azimuthally distributed such that they are each azimuthally separated by a 120° rotation about the long axis of the thick filament. Subsequent crowns within the 43 nm repeat are azimuthally offset by 60°, such that every cross-bridge faces a thin filament (Figure 3.1A) [Al-Khayat et al., 2008].

The lattice of thick and thin filaments is a hexagonally packed array. Lattice spacing within the model is isotropic, the radial distance between thick and thin filaments is the same in all orientations. The lattice spacing is an input parameter to the model and can be controlled independently of sarcomere length. The other radial distances, e.g. myosin plane separation (d_{10}) and actin plane separation, are determined by lattice geometry.

Availability of binding sites is modulated by thin filament overlap. When adjacent thin filaments overlap they shield each other's binding sites, disallowing the formation of cross-bridges at those locations. This simulates shielding of binding sites, consistent with the

classic understanding of reduced force on the ascending limb [Gordon et al., 1966].

3.5.3 Simulation details

A single 1 ms time-step in the model consists of allowing each myosin head to calculate the probability of a change in state and then check that probability against a random number between zero and one taken from a uniform distribution. The force at a given myosin base can be calculated from the locations of the base, the locations of the adjacent bases, and the actin location, if any, to which the myosin is attached (Figure 3.5C). After each myosin has switched or retained its state, the positions of each location along the thick and thin filaments is found such that the only locations experiencing a net force are those at the ends of the filaments. The sum of the force at the ends of each of the thick filaments is opposite and equal to the sum of the force at the ends of each of the thin filaments and is the force output of the model. Repeating this process generates the force traces from which maximum force is determined (Figure 3.5D).

For each combination of lattice spacing and sarcomere length, the maximum isometric force was calculated by taking the mean of the force developed in each of ten runs. The model was allowed to rise to a steady force level over 400 ms, or 400 time-steps at 1 ms resolution. The maximum force level was reached within the first quarter of the run for all parameter combinations, allowing the force of the run to be calculated as the mean of the instantaneous force at each of the last 50 time-steps.

Model runs were carried out in parallel on a dynamically created cluster of spot-priced machine instances in Amazon's EC2 service. Model runs were controlled with a first-in-first-out command queue hosted by Amazon's SQS.

3.5.4 Skinned moth flight muscle fiber bundles

Skinned-fiber experiments were carried out using the BioCAT beam-line 18ID at the Advanced Photon Source, Argonne National Laboratory [Fischetti et al., 2004]. Fibers were harvested from subunits B and C of the dorsolongitudinal flight muscles of *Manduca sexta* [Tu and Daniel, 2004]. The large fibers and well ordered structure of *Manduca* flight muscle

permit a strong diffraction pattern and stand up well to successive osmotic pressure changes. *Manduca* flight muscle also has a clear role in organismal movement and is easily available.

Individuals were selected fewer than five days post-eclosion, decapitated under cold anesthesia, and stored at 4°C for up to three days before their fibers were harvested. Fibers were harvested by bisecting the thoraces in the median plane and dissecting out fiber bundles in calcium-free relaxing solution [Kreutziger et al., 2007]. The fiber bundles were skinned overnight in a 1% Triton relaxing solution before being washed and stored for up to four days at 4°C in relaxing solution.

3.5.5 Apparatus

Skinned fiber bundles were mounted between two posts, the ends affixed with cyanoacrylate glue (Figure 3.6). The fiber was then lowered into a flow cell where the solution bathing the fiber could be changed by means of a multi-port programmable syringe pump (ML560C, Hamilton Company, Reno, NV). Fibers were mounted in pCa 9.0 relaxing solution and were activated by a two-step protocol. The pCa 9.0 solution was first replaced with an EGTA-free pre-activating solution before that was replaced with a pCa 4.0 activating solution. Activating solutions were made similarly to Kreutziger et al., with CaCl2 added to a desired Ca^{2+} concentration but omitting dithiothreitol, creatine kinase, and 2,3-Butanedione monoxime [Kreutziger et al., 2007]. Double washes at each solution swap ensured full exchange.

A dual-mode force lever (Model 305B, Aurora Scientific, Ontario, Canada) both held fiber-length constant and monitored force developed by the fiber. This force lever was coupled to one of the posts to which the fiber was mounted, while the other post was was rigidly held in place.

The flow cell was printed with ABS on a Dimension μPrint (Stratasys Inc, Eden Prairie, MN). Windows at the front and back of the flow-cell were covered with flat 2 mil Kapton sheets (McMaster-Carr, Atlanta, GA) to permit the X-ray beam to pass through the solution and fiber without scattering on the ABS. After passing through the fiber and exiting the flow cell, X-rays traveled through a further two meter evacuated flight tube before being imaged (example shown in Figure 3.4A) and used to measure the d_{10} lattice spacings as in

Irving, 2006 [Irving, 2006, Squire, 1983].

A diode laser mounted above the flow-cell was used to measure the average sarcomere length in the region being imaged with X-ray diffraction. A flat Kapton window below the fiber permitted the laser to pass through the cell and a strip of Kapton laid on the solution above the fiber eliminated distortion from solution meniscus. The center of the laser's primary diffraction line was found by eye and initial sarcomere length was stretched to 4.0±0.2 μm.

Contraction

Activating, relaxing, and pre-activating solutions were prepared with 0%, 2%, 4%, and 6% Dextran T500 by weight. All activations were carried out at 20°C. Prior to activation, fiber bundles were perfused with a pre-activating solution. Bundles were given 20 minutes to equilibrate to a new level of osmotic compression with each change in Dextran concentration. The force of each activation was taken as the difference between the peak force and the mean of the pre- and post-contraction relaxed force levels.

Analysis

Over the course of activations at 0% Dextran, two randomly chosen values from 2%, 4%, and 6% Dextran, and a final contraction at 0% rundown of the fiber bundles was observed. This rundown was compensated for by normalization of force to the I_{20}/I_{10} ratio. The I_{20}/I_{10} ratio is a measure of the radial distribution of cross-bridge electron density and is frequently used as a measure of the level of cross-bridges associated with the thin filament [Irving, 2006, Irving and Maughan, 2000]. This thus normalizes the force to the loss of functional cross-bridges that occurs over the course of multiple activations. Lattice spacing was normalized to the initial post-skinning lattice spacing to facilitate comparisons between skeletal and insect flight muscle.

Acknowledgements

Support was provided by an NSF grant to TLD and TCI, funds from the Joan and Richard Komen Endowed Chair to TLD, an NIH pre-doctoral training grant to CDW, an Amazon Grant For Research to CDW, and an NHLBI Project Grant to MR.

Use of the Advanced Photon Source, an Office of Science User Facility operated for the U.S. Department of Energy (DOE) Office of Science by Argonne National Laboratory, was supported by the U.S. DOE. Use of the BioCAT facility was supported by grants from the NIH.

Maria Razumova, Nicole George, and Simon Sponberg provided discussions and assistance. Nicole George and Wai Pang Chan also provided the EM image in Fig. 3.7. The authors would also like to thank Chen-Ching Yuan and David Gore for their assistance operating the BioCAT beamline.

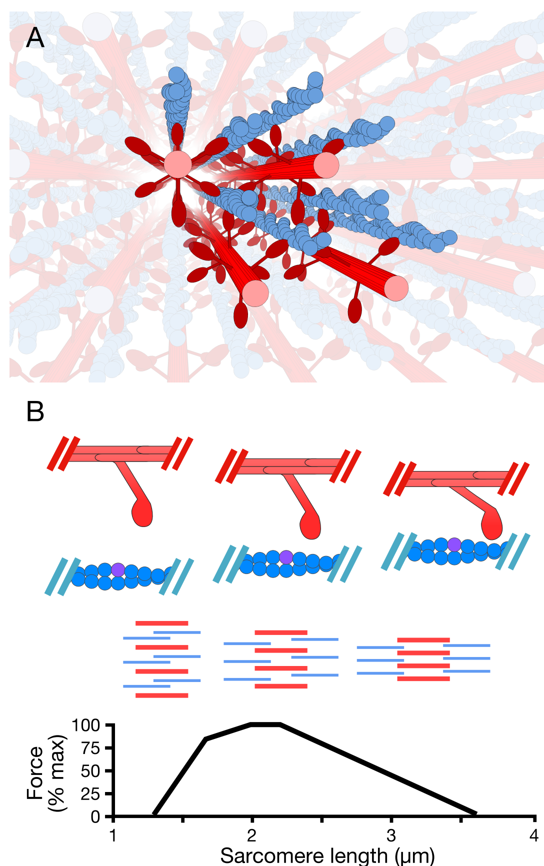
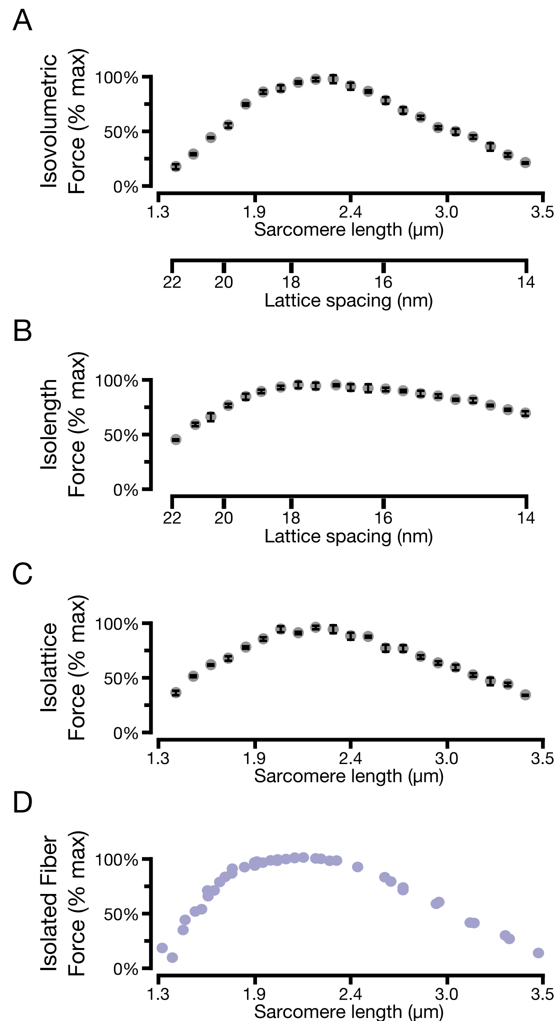


Figure 3.1: **A spatially explicit three-dimensional multifilament model of the sarcomere generates varying levels of force as lattice spacing varies across the length-tension (LT) curve A)** The multifilament model of the half-sarcomere uses eight thin filaments and four thick filaments, populated by two-dimensional cross-bridge models. Simulated filaments are depicted in bold colors, and mirrored versions of those filaments which are reached by passing through a boundary are shown as in light colors. Toroidal boundary conditions simulate an infinite lattice of contractile filaments which is sensitive to changes in radial spacing. **B)** Filament overlap and lattice spacing changes across the LT curve. From bottom to top, the LT curve for vertebrate skeletal muscle, a diagram of the relative degree of overlap and lattice spacing of the contractile filaments across the LT curve, and a cartoon depicting the relative locations of a single cross-bridge and the thin filament across the LT curve. In the filament schematic, the thick filaments (in red) are compressed together as overlap with the thin filaments (in blue) decreases from left to right. In the single cross-bridge cartoon, the lattice spacing decreases as sarcomere length grows from left to right. An example binding site is shown in purple, demonstrating in the rightmost column the geometric restriction that accompanies highly compressed lattice spacings.

Figure 3.2: **Isovolumetric, isolattice, and isolength force curves compared to isolated fiber measurements.** All sarcomere lengths and lattice spacings are chosen to reflect the range over which vertebrate striated muscle varies in isometric contractions along the length-tension (LT) curve [Millman, 1998]. Measured force from frog striated muscle at various sarcomere lengths is shown for comparison [Gordon et al., 1966]. **A)** The force generated along a simulated isovolumetric LT curve shows the classic three-region shape. To maintain a constant lattice volume, lattice spacing changes as the square root of 1/sarcomere length. Force at extreme sarcomere lengths/lattice spacings is decreased by over 75% from its peak value. **B)** The simulated LT curve where only lattice spacing changes, and sarcomere length is fixed at 2.4 μm, shows a reduced slope. However, force decreases by more than 50% from its peak at larger lattice spacings (corresponding to short sarcomere lengths) and decreases by more than 25% at smaller lattice spacings (corresponding to long sarcomere lengths). **C)** The simulated LT curve where only sarcomere length varies, and lattice spacing is held at 17.1 nm (corresponding to a d_{10} lattice spacing of 36nm), recreates more of the isovolumetric case's slope. Force decreases by over 50% at either extreme. **D)** For comparison to the above, the maximum force developed by isolated frog striated muscle is presented, modified from Gordon et al., 1966.



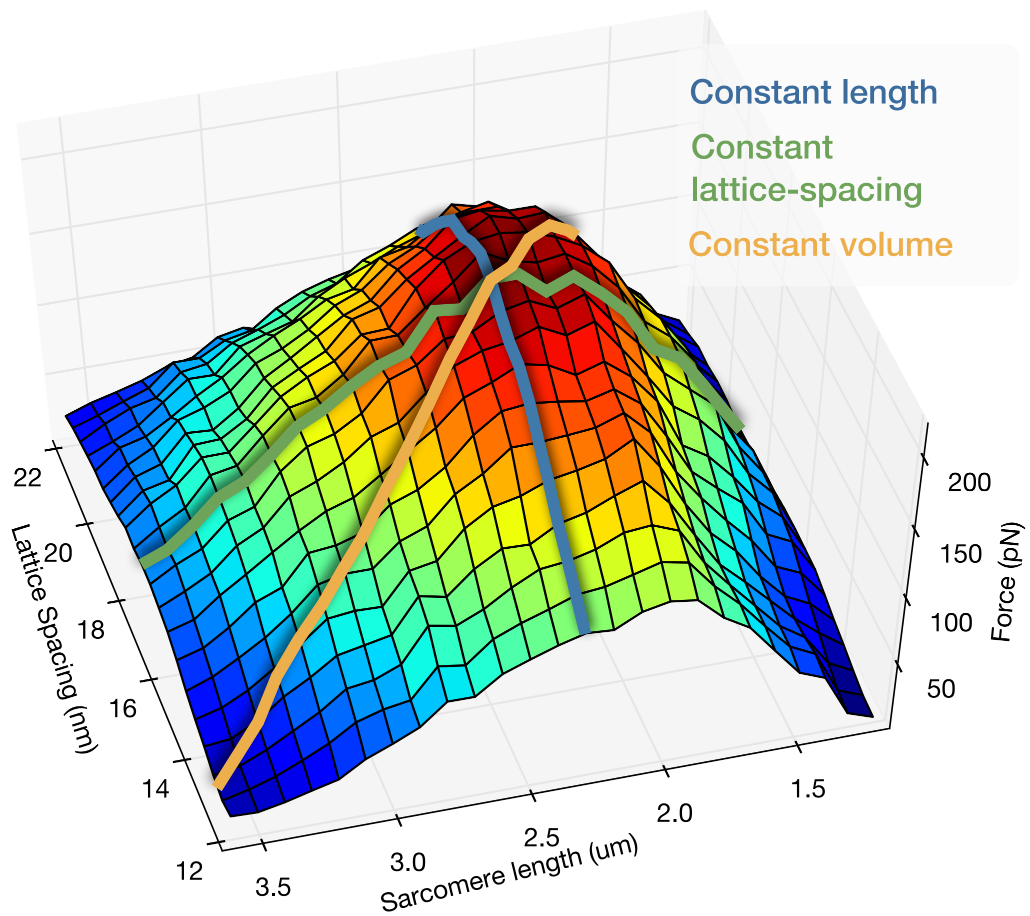


Figure 3.3: **Force at all lattice spacings and sarcomere lengths.** Both lattice spacing and sarcomere length tune force, as shown by the dependence of force on lattice spacing at all sarcomere lengths and on sarcomere length at all lattice spacings. An isovolumetric path through the parameters of lattice spacing and sarcomere length results in an LT curve of nearly maximal steepness. Peak force is seen at 17.5nm lattice spacing and 2.3μm sarcomere length.

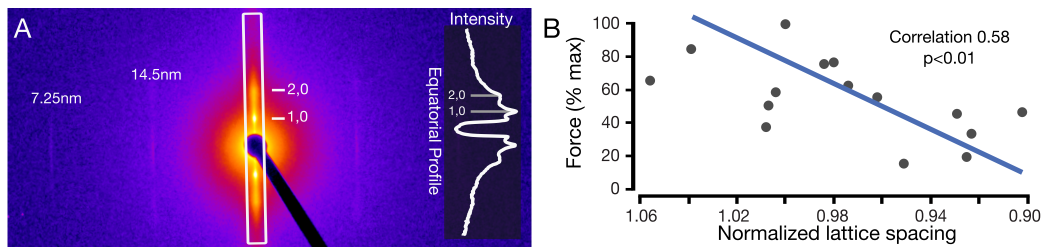


Figure 3.4: **Normalized force of skinned fibers under osmotic compression.** The force generated by skinned osmotically compressed fibers was measured using small-angle X-ray fiber diffraction, a sample image of which is shown. **A)** Diffraction image of skinned *Manduca sexta* flight muscle bundle with largely vertical equatorial axis. The dark line and dark central circle are due to a backstop protecting the detector from direct exposure to the X-ray beam. The 1,0 and 2,0 equatorial peaks, along with the 14.5 nm and 7.25 nm actin lines are labeled. The boxed intensity profile along the equator is inset. Lattice spacing and the I_{20}/I_{10} ratio were measured as in Irving, 2006. **B)** Maximum force produced through Ca^{2+} activation of fibers stretched to the descending limb decreases as lattice spacing is shrunk through osmotic compression. Lattice spacing, here the separation between adjacent myosin planes as measured by the d_{10} X-ray diffraction distance, is displayed as normalized to the lattice spacing on the initial post-skinning activation to control for variability between fibers.

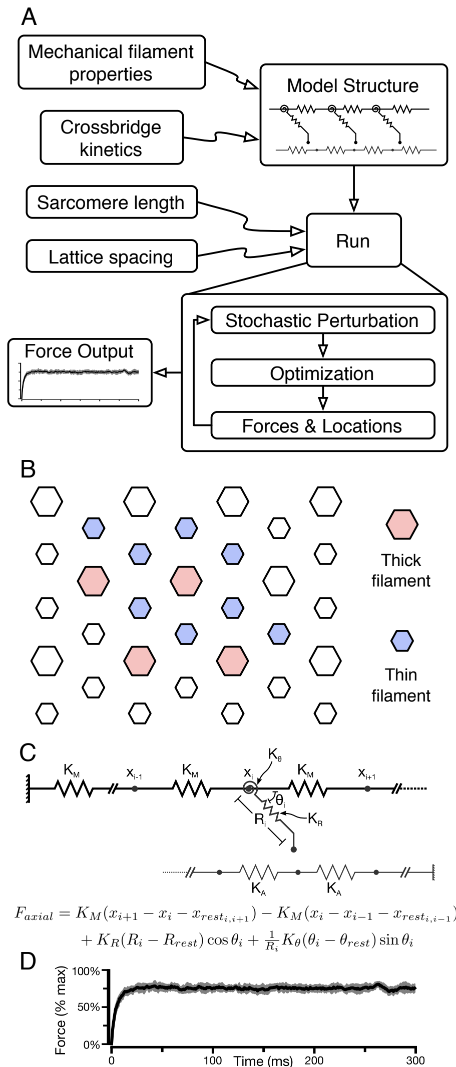


Figure 3.5: **Model architecture and filament geometry. A)** Diagrammatic representation of simulation steps and model property origins. The parameters of the model structure are derived from known mechanical and geometric properties of the contractile filaments, as well as myosin kinetics. For each run, the sarcomere length and lattice spacing are given as input parameters. **B)** The arrangement of the thick and thin filaments in the model is such that four thick and eight thin filaments are sufficient to replicate a semi-infinite lattice. The colored filaments are those simulated. The white filaments are reflections of the simulated filaments, reachable by wrapping through one of the boundaries. This lattice arrangement is drawn from vertebrate lattice measurements [Millman, 1998, Tanner et al., 2007]. Our experimental data is drawn from insect flight muscle. Insect flight muscle has a different lattice arrangement, as depicted in Figure 3.7B&C, but observes the same lattice spacing scaling rules as the vertebrate lattice under isovolumetric conditions. **C)** Schematic representation of the springs surrounding a single cross-bridge location. The equation which yields the axial force felt at the base of the myosin head is shown below. This force is dependent on the relative locations of adjacent myosin attachment points and any actin site to which the myosin head is bound. Axial force is balanced at all internal locations, such that net force is only experienced at the ends of the modeled filaments. **D)** The force traces seen at a representative lattice spacing and actin-myosin filament overlap show the development of force within the model. Mean (black line) and standard deviation (gray region) of 10 force traces at 15.5 nm lattice spacing and 2.5 μm sarcomere length. The force is depicted as a percentage of the maximum force the model generates in an isovolumetric length-tension curve (218 pN).

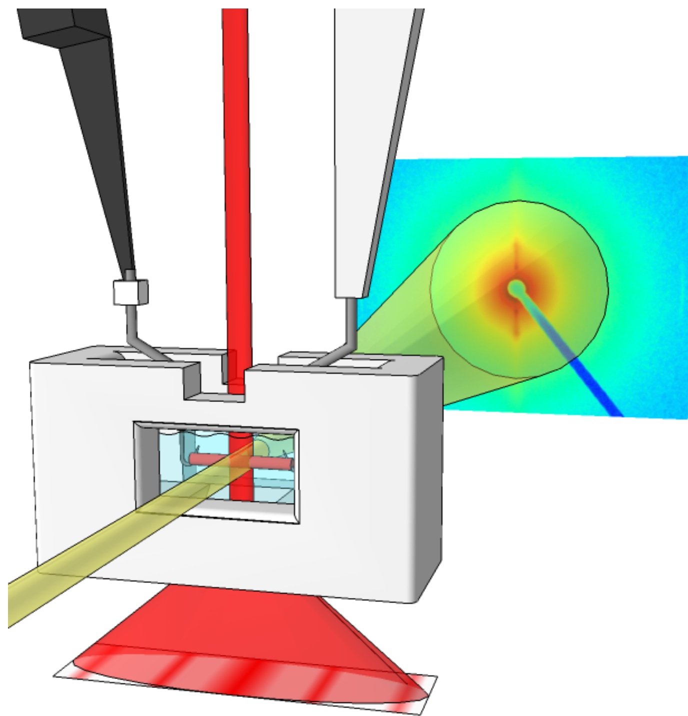


Figure 3.6: **Skinned fiber bundle apparatus.** Schematic of skinned fiber bundle physiology apparatus used at the BioCAT beamline 18ID at the Advanced Photon Source to measure the force and lattice spacing of fibers while controlling their activation, osmotic compression, and length. The bundle was attached to two hooks, one of which was connected to a dual mode force lever. Activation and osmotic compression were controlled through immersion in the solution of a flow chamber. Sarcomere length was measured through laser diffraction. Lattice spacing was measured using synchrotron X-ray fiber diffraction.

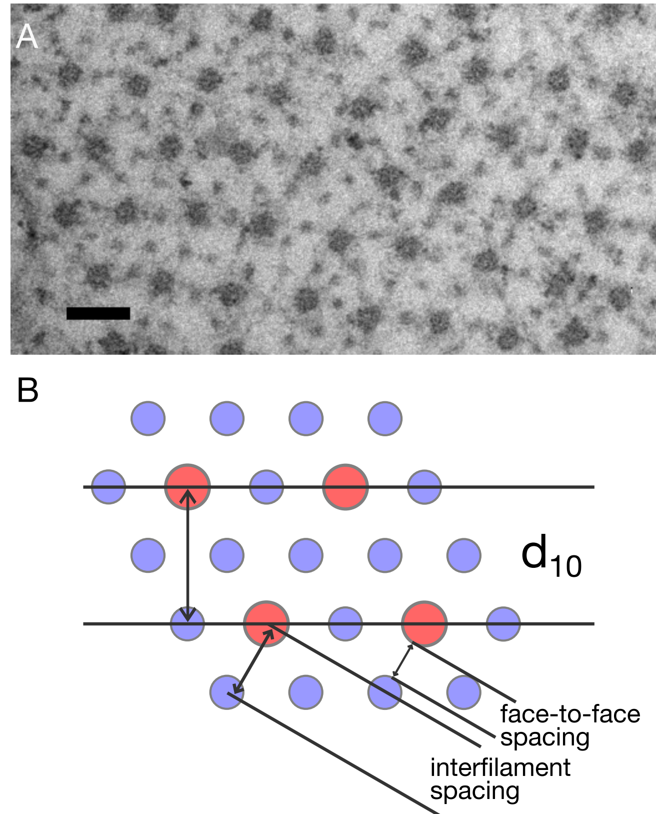


Figure 3.7: **Lattice arrangement.** The lattice structure of *Manduca sexta* flight muscle is similar to that of other insect flight muscle. **A)** The lattice of *Manduca sexta* flight muscle, as seen in this electron microscopy image of a transverse section of dorsolongitudinal muscle, is a hexagonal structure similar to that of asynchronous flight muscle. A scale bar of 50nm is shown and the image was taken at 92,000x magnification. **B)** The d_{10} lattice spacing measured with X-ray diffraction is converted to the face-to-face lattice spacing. Our modeling work uses the spacing between adjacent filament faces as the lattice spacing [Williams et al., 2010]. The interfilament and d_{10} spacings are also often referred to as lattice spacing. The d_{10} and interfilament distances are directly convertible, and both may be converted to the face-to-face spacing given filament thicknesses.

Chapter 4

ELASTIC ENERGY STORAGE AND RADIAL FORCES IN THE MYOFILAMENT LATTICE DEPEND ON SARCOMERE LENGTH

C David Williams, Michael Regnier, Thomas L Daniel

4.1 Abstract

We most often consider muscle as a motor generating force in the direction of shortening, but have recently come to recognize that it also fills roles as a spring or a brake. Here we develop a fully three-dimensional spatially explicit model of muscle to isolate the locations of forces and energies that are difficult to separate experimentally. We show that the strain energy in the thick and thin filaments is less than one third the strain energy in the attached cross-bridges. This result suggests the cross-bridges act as springs, storing energy within muscle in addition to generating the force which powers muscle. Comparing the energy consumed by the model to the elastic energy stored by the model, we show that the ratio of these two properties changes with sarcomere length; the model stores a greater fraction of energy at short sarcomere lengths. Additionally, we investigate the force that muscle produces in the radial or transverse direction, orthogonal to the direction of shortening. We confirm prior experimental estimates that place radial forces on the same order of magnitude as axial forces, although we find that radial forces and axial forces vary differently with changes in sarcomere length.

4.2 Author Summary

Locomotion requires energy. Very fast locomotion requires a larger amount of energy than muscle can produce in such a short time period, and so it must use energy that we previously produced and stored as elastic deformation. Traditionally we've looked at tendons, insect exoskeletons, and bones to find the locations where this energy is stored, but a small body of literature has suggested that the backbone filament proteins in muscle act as elastic storage

locations. We suggest that the myosin motors on the filament proteins are actually storing more energy than the filaments themselves, energy which may be released to power very fast movements. We further suggest that this energy is being stored as a result of deformations out of the axis in which muscle shortens.

4.3 Introduction

4.3.1 Energy storage in cross-bridges

Strain energy storage in muscle systems is most often associated with stretched tendons or other elastic supporting materials [Alexander, 2002, Gosline et al., 2002]. In many instances, strain energy storage in skeletal and tendon structures has been shown to be a crucial component of the locomotor systems of animals, especially flying animals [Ellington, 1984]. While muscle's role as a force generator has dominated research on animal locomotion, there are emerging studies that posit diverse functional roles for muscles, including those of a brake, actuator, spring, or even a damper (for a review see Dickinson et al. [2000]). Somewhat less attention has focused on the extent to which muscle itself plays a role in strain energy storage. That work which has been done has focused on the possibility of storing energy in the thick filaments, rigored cross-bridges, or the in the extensible accessory protein titin [Tidball and Daniel, 1986, Dickinson et al., 2005, Monroy et al., 2007]. This assumption that active cross-bridges play a minor role is understandable: they generate force in activated muscle and are thought to be constantly cycling between freely diffusing and attached states and so would be expected to develop little deformation.

However, recent work suggests that in certain situations the cross-bridges may be locked onto muscle's thin filaments, frozen into a lattice that can act to store energy [George et al., 2012]. Energy storage may be possible in the subset of bound cross-bridges in antagonistic muscles that absorb inertial energy of a periodically moved appendage. Additionally, energy storage in muscle has been proposed in non-cyclical movements such as the tentacular strike of the squid, stomatopods' raptorial appendage strike, or the tongue extension of toads [Kier and Leeuwen, 1997, Zack et al., 2009, Lappin et al., 2006, Patek et al., 2011]. In these cases of one-off sudden movement, even a set of cycling cross-bridges may store strain

energy for release on the initiation of rapid movement through pre-movement activation and subsequent pre-movement strain of the cross-bridges occurring just before the onset of an explosive motion. A new spatially explicit model lets us parse how strain energy is partitioned between the filaments and the cross-bridges in maximally activated isometric sarcomeres. We show that the cross-bridges may store the majority of the elastic strain energy.

4.3.2 Force generation in the radial direction

Cross-bridges are more often thought of as force generators than energy storers. The force generated by cross-bridges arises from deformations of individual myosin heads as they form cross-bridges between the thick and thin filaments and undergo a rotation about a lever arm [Spudich, 2001]. Interestingly, generating force by a rotation about a hinge implies that the vector of the generated force will have a component perpendicular to the direction of contraction [Schoenberg, 1980b, Williams et al., 2010]. This force component is in the radial direction, orthogonal to the axial force that is generated in parallel to the thick and thin filaments (Figure 4.1).

Radial force was observed during contraction in intact muscle fiber experiments dating back to the 1950s [Hiramoto, 1956]. Subsequent studies of radial forces placed them on the same order of magnitude as axial force [Maughan and Godt, 1981b, Matsubara et al., 1984, Cecchi et al., 1990, Brenner and Yu, 1991b, Xu et al., 1993, Nyland and Maughan, 2000]. These more recent experiments addressed radial force production through a proxy such as changes in fiber diameter or alterations of the muscle's radial compliance. The use of a lever-arm cross-bridge in the current spatially explicit model permits direct simulation of radial force production (Figure 4.1B&D) [Williams et al., 2010]. This cross-bridge model expands upon prior models (Figure 4.1A&C) [Schoenberg, 1980b, Pate and Cooke, 1989, Daniel et al., 1998, Tanner et al., 2007, Campbell, 2009].

Radial force may have functional implications. The internally generated radial force is a partial determinant of fiber radial compliance [Xu et al., 1993, Nyland and Maughan, 2000]. Alterations in radial fiber compliance are also a hallmark of dystrophic disorders [Pasternak

et al., 1995, Batchelor and Winder, 2006]. Misregulation of the transmission of radial force produced during contraction may be a cause of the disorder observed in histological studies of dystrophic muscle [Blake et al., 2002].

In addition to the more commonly analyzed axial forces, the model presented here addresses both radial force generation and the strain energy in the filaments and cross-bridges of the contractile lattice. These phenomena are linked, and are results of deformation of cross-bridges in the axial and radial directions. The interdependence of these properties is uniquely addressable using spatially explicit models of muscle contraction with lever-arm myosin geometries we have developed based on protein structural information [Williams et al., 2010]. Such models permit a fine parsing of energy locations which shows that cross-bridges store substantial elastic strain energy. Correlation of this cross-bridge energy with axial and radial forces suggests that radial cross-bridge strain could supply much of the energy stored in the contractile lattice.

4.4 Results

Radial forces produced within the half sarcomere are both large and correlated with energy storage. Our model monitors radial forces produced by lever arm cross-bridge models composed of an angular and an extensional spring (Figure 4.1A&B). The forces and energies used for comparison are steady state values produced on full isometric activation at a range of sarcomere lengths stretching across the length-tension curve. Even though isometric contraction represents only one possible loading regime, it is the most logical condition in which we can explore how sarcomere length, and thus filament overlap influences axial and radial forces with two-dimension models of force generation by cross-bridges.

4.4.1 Axial and radial forces are of the same order of magnitude

In the fully activated conditions of our simulations, both the axial and radial forces quickly rise to an asymptotic maximum (Figure 4.2). This rise to a maximum value takes less than 50 ms. The exponential time constant of the rise to peak force is not significantly different between the axial and radial forces ($p=0.31$). After steady force levels are reached,

stochastic fluctuations in the number and states of bound cross-bridges show as noise in the force traces. This clean rise gives clear asymptotic maximum forces.

The radial force, at all sarcomere lengths, is of the same order of magnitude as the axial force (Figure 4.3). At most overlaps, i.e. at sarcomere lengths below 3.0 μm, the radial force is larger than the axial force. At very short sarcomere lengths the radial force is as much as 2.4 times the axial force, although this is in a region where overall axial force levels are relatively small. These results agree with prior experimental studies which found radial forces of the same magnitude as axial forces [Maughan and Godt, 1981b, Matsubara et al., 1984, Cecchi et al., 1990, Brenner and Yu, 1991b, Xu et al., 1993, Nyland and Maughan, 2000].

4.4.2 The cross-bridges store the majority of strain induced energy

The majority of strain energy stored in the contractile lattice of filaments and cross-bridges is partitioned in the cross-bridges (Figure 4.4A&B). The ratio of the energy in the cross-bridges to that in the filaments varies with sarcomere length (Figure 4.4B). Even when the filaments reach their peak energy relative to the cross-bridges, at the sarcomere lengths where maximum force is produced, the cross-bridges still have more than three times the energy of the filaments. At very long and very short sarcomere lengths, where little axial force is produced, the energy stored in the cross-bridges dominates the system.

The elastic energy storage may be more finely parsed: into the components located in each of the two springs constituting a cross-bridge and each of the two filament types (Figure 4.6). This shows the energies of the thick and thin filaments are similar across all sarcomere lengths. In contrast, the energies of the torsional and extensional spring which comprise the cross-bridge are quite different. The energy of the torsional spring is far less than that of the extensional spring at all but the smallest sarcomere lengths. This suggests the majority of the elastic strain felt by the cross-bridge may arise from stretching, rather than rotation.

4.4.3 Cross-bridge energy correlates with radial force

Energy stored in the cross-bridges follows the radial force produced by system (Figure 4.4C). Radial force has a higher correlation with the cross-bridges' energy than does axial force (linear fits show respective r^2 values of 0.97 and 0.69). All of these relationships are significant (p¡0.001). This suggests that radial strain in the system may disproportionately determine the energy stored in the cross-bridges or, put another way, radial deformation may be acting as a hidden energy sink.

4.4.4 Fractional energy stored is not constant

Below sarcomere lengths of 2.0 μm the energy stored in the sarcomere relative to the energy consumed by the system increases (Figure 4.5). This fraction of energy stored, or energy retention efficiency, is constant at sarcomere lengths longer than 2.0 μm. The hydrolysis of ATP to ADP alters the free energy a modeled cross-bridge by 8.8 RT [Tanner et al., 2007, Williams et al., 2010]. A fraction of this energy is stored as continued deformation of the cross-bridge in its new state and a fraction becomes deformation of the filaments the cross-bridge exerts force on. This energy is entirely dissipated on the detachment of the cross-bridge and may be partially dissipated by deformations induced by other bound cross-bridges. At sarcomere lengths longer than 2.0μm the ratio of the energy stored by the sarcomere to the power consumed by the sarcomere (as measured by the rate of ATP consumption) is constant (Figure 4.5). At shorter sarcomere lengths this ratio climbs: more of the input energy is stored in the sarcomere.

4.5 Discussion

The role that muscle's radial geometry plays in determining its functioning is still poorly understood. It is difficult to experimentally measure the forces muscle generates in the radial direction and the strain and energies which result from such forces. The studies that have attempted to measure radial forces have all done so indirectly, through back-calculating from changes in radial stiffness or lattice spacing changes on activation [Cecchi et al., 1990, Brenner and Yu, 1991b, Nyland and Maughan, 2000]. While these are easier values to

quantify, they are not direct measurements of radial force. Our results suggest radial forces, in addition to being quite large, may function as a "hidden" energy storage mechanism, that radial force may partition energy into the cross-bridges which is not initially transmitted to the filament ends.

4.5.1 Substantial energy is stored in muscle's contractile elements

The elements of the sarcomere's contractile lattice, cross-bridges as well as thick and filaments, are storing a substantial amount of energy. This strain energy is primarily stored in the cross-bridges, rather than in the thick and thin filaments. Energy storage in the cross-bridges requires low cross-bridge turnover, as the deformation of an individual cross-bridge, and thus the energy in an individual cross-bridge, dissipates upon detachment. This locked lattice of cross-bridges is likely to be present in a maximally activated isometric contraction, as simulated here, and where external factors such as temperature differentials reduce cross-bridge turnover [George et al., 2012]. This suggests that muscle may be able to achieve spring-like operation in more than the sense of producing a force proportionate to its length. Muscle which stores energy in the filaments and cross-bridges may absorb energy for later release and utilization.

4.5.2 Radial force may be a 'hidden' means of storing energy

Radial force's role in muscle remains unclear. Radial force may simply be a byproduct of the motor and filament geometry which has evolved to generate force or it may produce a useful effect. The high correlation between radial force and strain energy stored in the cross-bridges may indicate that radial force and distortion acts as an energy storage mechanism which permits the cross-bridges to store more strain based energy than the thick and thin filaments. It is possible that radially associated energy could then be redirected to produce axial force, much as happens when energy is stored in the deformation of elastic solids. Radial strain based energy storage will not necessarily register as force at the filament ends and so may be difficult to address in experiment, although radial stiffness observations suggest a means by which such tension and energy storage could be quantified [Nyland and

Maughan, 2000].

4.5.3 The fraction of energy stored varies with muscle length

The variable energy retention efficiencies shown in Figure 4.5 represent a potential mechanism by which sarcomeric parameters can determine a muscle's functional role, e.g. motor, brake, or spring. A muscle that stores little of its consumed energy and converts it into force acts as a motor. While a muscle that stores more of its consumed energy for later release is acting, at least temporarily, as a spring. There are analogous selections between roles in lengthening and shortening muscle, as contrasted to the isometric conditions simulated here [Dickinson et al., 2000].

The non-constant storage of input energy means that changing the degree of filament overlap or lattice spacing affects the amount of energy stored in muscle and thus the amount of energy which may be released to power contraction. Thus it is possible that operating at different sarcomere lengths could change the efficiency with which muscle retains energy or dissipates it, driving the muscle towards functioning as a spring or a break.

Energy storage in muscle is still a relatively unexplored field. The highly structured and three-dimensional nature of muscle makes it likely that, historically, we have overlooked forms of energy storage and efficiency regulation. This work points towards cross-bridges as a site where energy can be stored in preparation for single-use release. Further investigation of how, in cyclical moments, energy is partitioned and the levels of radial forces generated will continue to expand our understanding of these new mechanisms. This work may help us to understand cases where energy is stored out of the plane of use, as in proposed mechanism by which the heart stores elastic strain introduced into transverse fibers during filling [Torrent-Guasp et al., 2005].

4.6 Methods

Our spatially-explicit model of the half-sarcomere represents the thick and thin filaments as chains of springs connecting each myosin crown or actin-binding site [Tanner et al., 2007, Daniel et al., 1998]. The cross-bridges linking the contractile filaments are two-dimensional, and thus both produce radial force and are sensitive to changes in lattice spacing [Williams

et al., 2010]. This is the only model we are aware of that accounts for the lattice spacing between filaments and employs a cross-bridge capable of simulating force in the radial direction.

4.6.1 Thick/thin filament arrangement and geometry

Four thick and eight thin filaments are arranged in an evenly spaced hexagonal lattice with toroidal boundary conditions. This filament arrangement simulates an infinite lattice [Tanner et al., 2007]. The arrangement of boundary conditions is such that no single thick filament connects to two sides of a single thin filament, and vice versa. The distance between the faces of these filaments, here referred to as the lattice spacing, is uniform and used to provide the distance across which myosin must diffuse in order to bind [Williams et al., 2010].

Along each thick filament are 60 myosin crowns, with three myosin heads per crown. The myosin heads on a given crown are azimuthally rotated by 120 degrees from their neighbors. The crowns are grouped into a three crown, 43 nm repeating pattern [Tanner et al., 2007]. Progressing axially through the pattern, each crown is azimuthally rotated relative to the prior crown by 60°. This rotation pattern is measured and described in Al-Khayat et al. [2008]. As a result of this crown rotation every myosin head faces an opposite a thin filament with which it may interact.

Each thin filament is made up of two actin strands. Each strand hosts 45 actin binding-sites giving a whole filament 90 actin binding-sites [Egelman, 1985, Daniel et al., 1998, Tanner et al., 2007]. Each binding site faces and interacts with one of three adjacent thick filaments. Consecutive binding sites on each strand are rotated by 120°clockwise. The first binding site of one strand is offset from the first binding site of the other strand by half the axial distance between adjacent binding sites and a rotation of 120°counter-clockwise.

4.6.2 Cross-bridge model, briefly

Our cross-bridges are comprised of one torsional spring and one extensional spring [Williams et al., 2010]. The axial and radial locations of the myosin's tip determines the angle and

extension of the cross-bridge's springs and thus the force the cross-bridge generates. The torsional spring simulates the power stroke via a change in rest angle.

The binding of an individual myosin head is determined by the distance to the nearest available binding site and energy landscape created by the properties of the head's constituent springs. The process is one of perturbation, distance calculation, and stochastic attachment. A myosin head is perturbed with a random Boltzmann distributed energy, providing a new myosin tip location [Dill and Bromberg, 2002]. Distance from the myosin tip to the nearest available binding site is calculated. Binding probability, which falls off exponentially as the distance to the binding site increases, is checked against a random number. Further transitions between loosely attached, force generating, and unattached states are determined as described in Williams et al. [2010].

4.6.3 Force transmission through the lattice - calculation of axial and radial forces

The thick and thin filaments are coupled together by the cross-bridges. Each bound cross-bridge both generates and transmits force. This coupling yields a three dimensional network of springs.

We solve for the root location of our spring-network at each time-step. The root is the set of locations of actin binding-sites and myosin crowns that provides no net axial force at any internal point in the spring-network. A modified form of the Powell hybrid method allows the actin and myosin locations to iteratively settle into their solution values [Jones et al., 2001].

At each time-step, actin and myosin locations are allowed to settle in the axial dimension while being held rigidly in the radial dimension. The total axial force (F_{ax}) of the system thus comes as the sum of unbalanced axial forces at the ends of each thick filament,

$$F_{ax} = \sum_{i=1}^{N_{thick}} k(x_0 - x_{1,i} - d_{rest,i})$$

where for a given thick filament, i, k is the filament stiffness, x_0 is the axial location of the end node location, $x_{1,i}$ is the axial location of the adjacent node, and $d_{rest,i}$ is the resting separation between the two. The total radial force (F_{ra}) is found as the sum of the

radial force experienced by each of the thick filaments' faces and thus ultimately the sum of the radial force of each cross-bridge (XB),

$$F_{ra} \sum_{i=1}^{N_{XB}} k_\rho(\rho_i - \rho_{rest,i}) \sin\theta_i + \frac{k_\theta}{\rho_i}(\theta_i - \theta_{rest,i}) \cos\theta_i$$

where k_ρ, ρ_i, and $\rho_{rest,i}$ are the stiffness, length, and rest length of cross-bridge i's extensional spring while k_θ, θ_i, and $\theta_{rest,i}$ are the stiffness, angle, and rest angle of cross-bridge i's torsional spring. The current model does not permit radial movement as it does not include a resistive radial force, i.e. forces in the radial direction that act against the radial deformation of the filament. Future models may treat the filaments as radially-deformable axially-tensioned beams subject to filament persistence length and electrostatic effects and thus be able to permit radial movement.

4.6.4 Energies of a filament or cross-bridge is the sum of its springs' energies

The energy in a cross-bridge or filament is the sum of the energy in every spring in that cross-bridge or filament. Thus the energy of a single cross-bridge ($U_{\theta,\rho}$) is calculated, as in prior work [Williams et al., 2010].

$$U_{\theta,\rho} = \alpha\Delta G + \frac{1}{2}(k_\rho(\rho - \rho_{rest})^2 + k_\theta(\theta - \theta_{rest})^2)$$

The energy of a thick filament (U_{thick}) with N_{nodes} crown locations is calculated as

$$U_{thick} = \frac{1}{2} \sum_{i=1}^{N_{nodes}} k(x_{i-1} - x_i - d_{rest,i})^2$$

and the energy of a thin filament is calculated similarly. These energies are logged throughout the run of an instance of the model and their stable state is found at the conclusion of the simulation.

4.6.5 Simulation details

A simulated contraction follows the course described in the diagram shown in Figure 4.7. Briefly, each 1 ms time-step consists of allowing every myosin head to calculate the proba-

bility of changing from its current state into another state, check this probability against a random number, and transition or not based on the outcome. After the state of each myosin head has been established, the locations of every interior point in the model is allowed to settle so that there is no next axial force on them. The axial force, radial force, and other properties of the system at that time-step are then recorded and a new time-step is begun if the contraction has not yet reached its end.

The model was allowed to complete 10 contractions (starting from unbound cross-bridges) for every set of input parameters, each continuing for 400 ms (400 time-steps at 1 ms resolution). The asymptotically developed forces and energies were calculated as the mean of the force produced over the last 50 ms.

These simulations took place on a dynamically created cluster of spot-priced machine instances in Amazon's EC2 service (Figure 4.8). Control of this cluster was with a first-in-first-out command queue hosted by Amazon's SQS.

4.7 Acknowledgements

Funding was provided by an NIH pre-doctoral training grant to CDW, an Amazon Grant For Research to CDW, and funds from the Joan and Richard Komen Endowed Chair to TLD.

The authors would like to thank Simon Sponberg for statistical advice and Nicole George for helpful discussion.

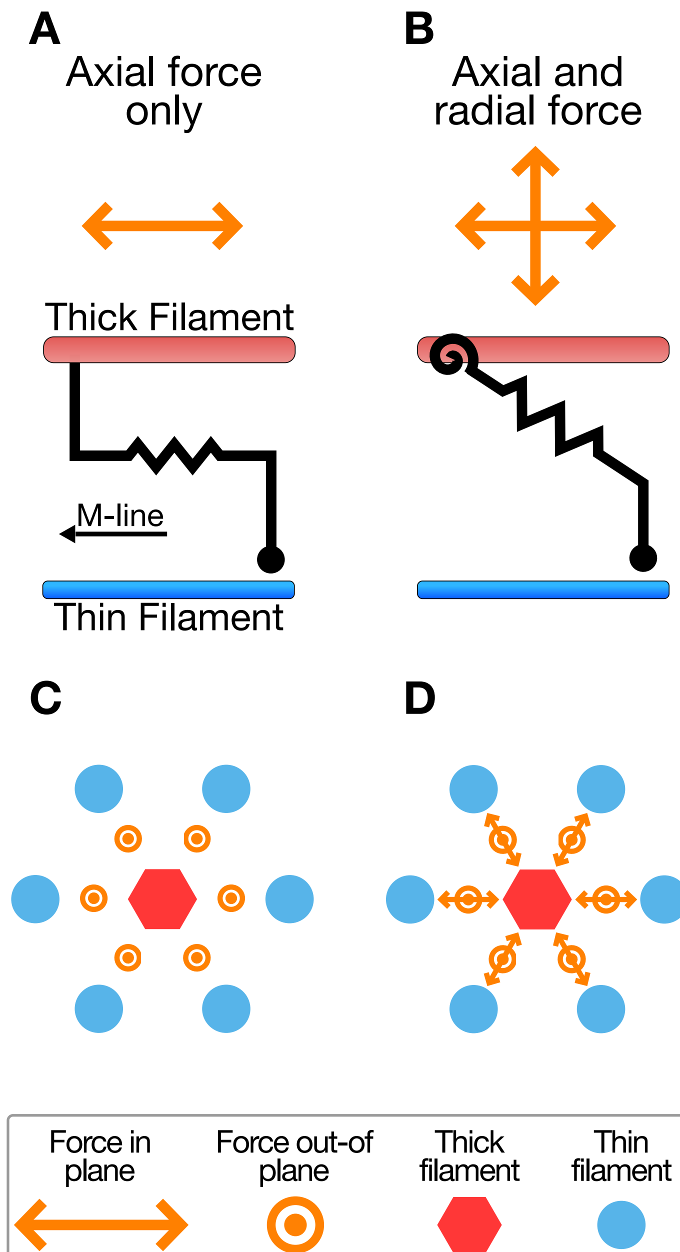


Figure 4.1: **Models produce radial and axial forces.** The one-dimensional cross-bridge model shown in **A** produces force and exists only in the axial direction. The two-dimensional cross-bridge model shown in **B** produces both axial and radial forces, and responds to changes in lattice spacing. A multi-filament model using one-dimensional cross-bridge, shown in **C**, is diagrammed as a three-dimensional system but is insensitive to changes in lattice spacing and unable to explore radial force produced during contraction. Using two-dimensional cross-bridges in the same model geometry, in **D**, allows the recording of radial forces and altered force dynamics with altered lattice spacing.

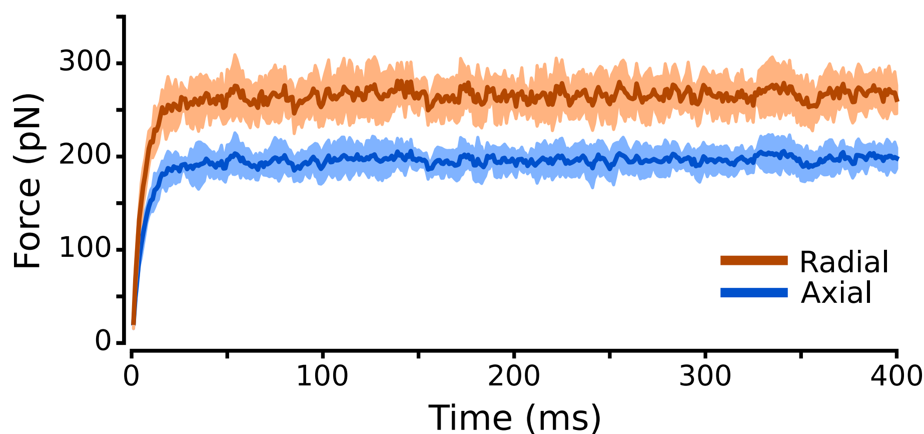


Figure 4.2: **Example axial and radial forces.** The mean (lines) and standard deviations (shaded regions) of axial and radial forces as they develop at a sarcomere length of 2.5 μm over the course of 10 runs. Each run consists of 400 time steps, each 1ms long. Maximum forces are calculated from the mean of the last 100 ms of such runs.

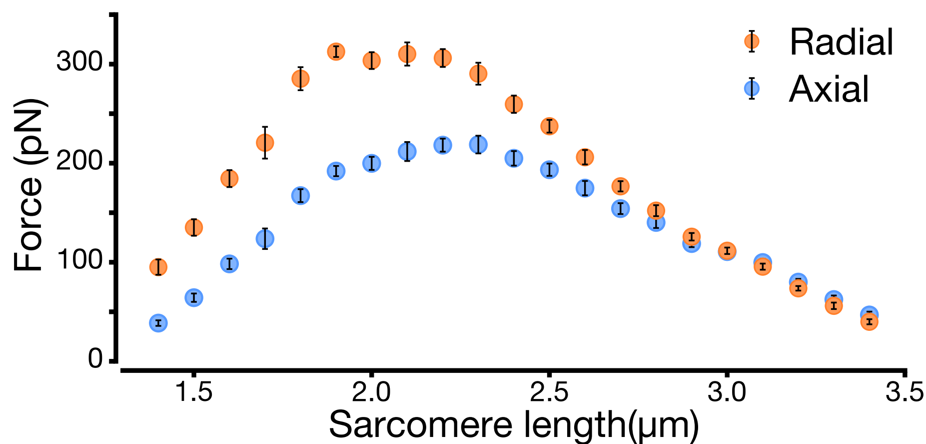


Figure 4.3: **Radial force is of the same order of magnitude as axial force.** Asymptotic maxima of 10 runs at each sarcomere length with standard deviation. Radial and axial forces obey similar scaling trends across the sarcomere lengths and lattice spacings of a classic length-tension curve. The level of radial force varies from 2.4 times the level of axial force at extremely short sarcomere lengths to 0.9 times the axial force at the longest sarcomere lengths. The radial force plateau ends at a shorter sarcomere length than does axial force plateau.

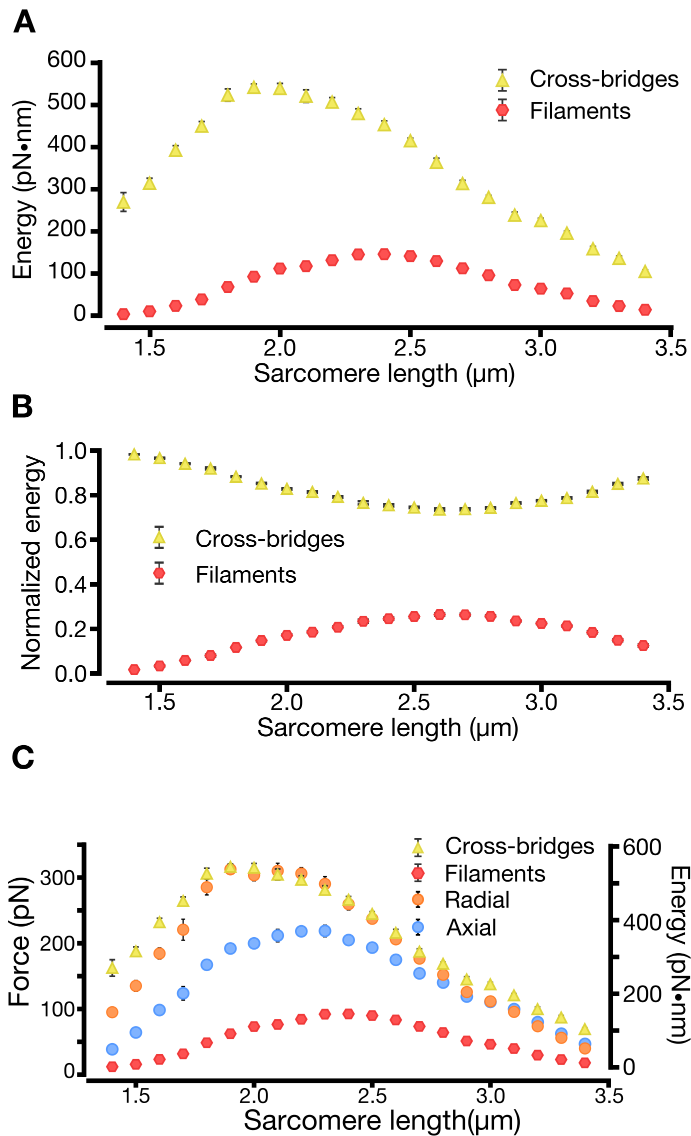


Figure 4.4: **Energy is partitioned between the filament backbone and the cross-bridges.** The energy stored in the springs comprising the cross-bridges and filaments changes, much as force does, with sarcomere length. As sarcomere length increases, the energy stored in the cross-bridges rises more steeply than does the energy stored in the energy stored in the filaments.

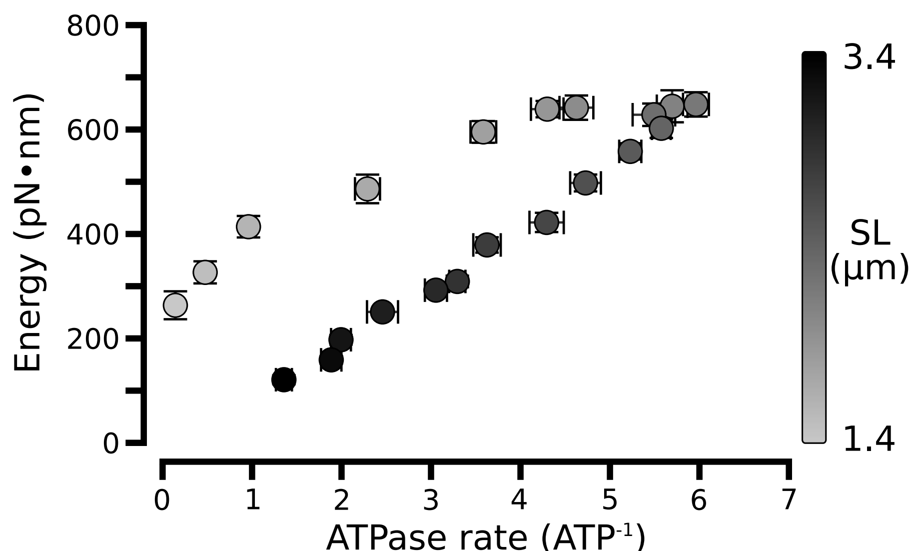


Figure 4.5: **System energy varies with sarcomere length as well as energy input.** The input of energy into the contractile lattice via the consumption of ATP determines the energy stored in the filaments and cross-bridges, but in concert with sarcomere length. A contractile lattice with an energy dependent only on the rate at which ATP is consumed would not exhibit the hysteresis present as sarcomere length changes. This suggests that of the energy released by ATP, the fraction which is stored instead of being dissipated is partially determined by sarcomere length.

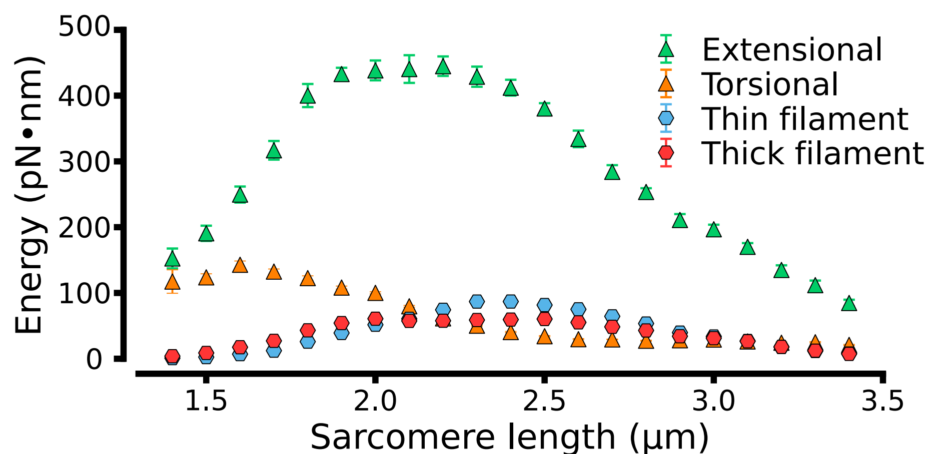


Figure 4.6: **Energy partitioned by filament type and cross-bridge spring.** The energy stored in the thick and thin filaments is approximately equal, while the extensional spring of the cross-bridges stores the major share of the energy at all sarcomere lengths.

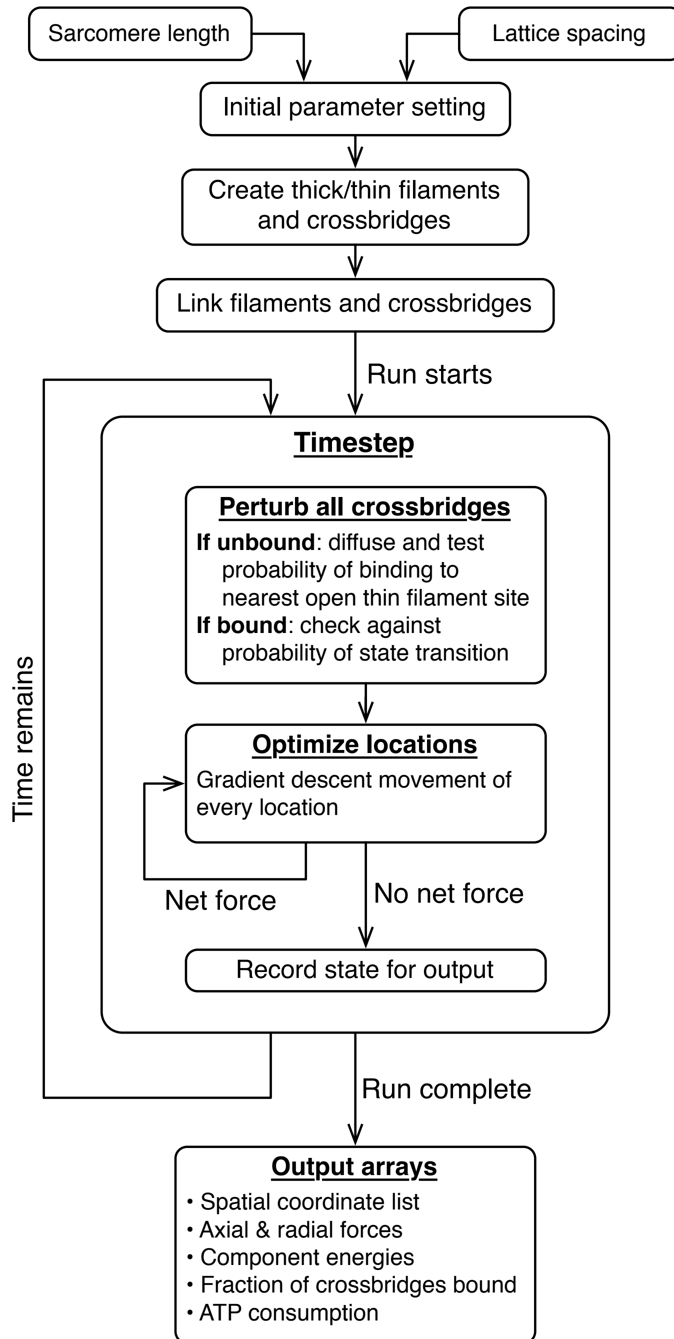


Figure 4.7: **Model code structure and information flow.** A diagrammatic representation of the steps that occur during a simulation, and which produce the measured forces and energies.

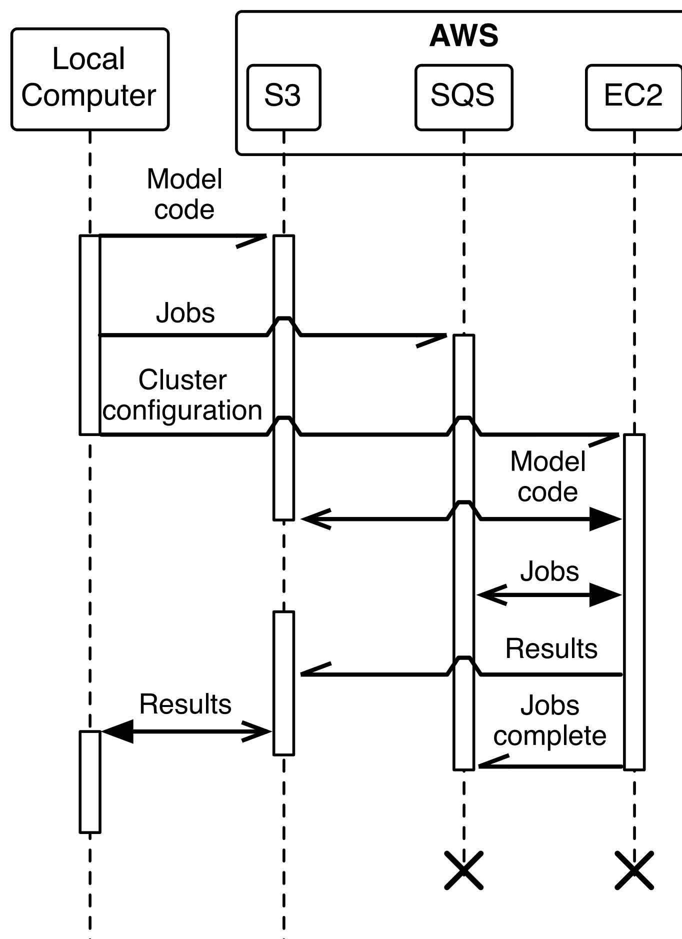


Figure 4.8: **Sequence diagram of remote simulation process.** This sequence depicts the process of running and retrieving results from Amazon's Web Services (AWS). Three AWS services are used: the Simple Storage Service (S3), the Simple Queue Service (SQS), and the Elastic Compute Cloud (EC2). The custom Python code which constitutes the model is sent to and stored in S3. The parameters which describe simulations are parsed into small jobs, which are sent to SQS. The remote cluster which will run the simulation is configured on EC2. When machine instances which make up the cluster are allocated and have started, they connect to S3 and download a copy of the model. They then connect to SQS and request jobs to run on each of their available cores. As jobs are completed, their results are uploaded to S3 and the completed job is removed from the SQS queue. The machine instances of the cluster will continue this process until the job queue is empty at which point they shut themselves down. The complete result set may then be downloaded for local processing, or processed by another EC2 cluster, as needed.

Table 4.1: **Geometric and mechanical properties** A listing of the geometric and mechanical parameters used in the model. All values are given for the half-sarcomere model and thus refer to one half of a thick or a thin filament.

Filament	Parameter	Value	Source
Thick filament	Rest length	858 nm	[Tanner et al., 2007]
	Undecorated region length	80 nm	[Higuchi et al., 1995]
	Inter-crown spacing	14.3 nm	[Tanner et al., 2007]
	Number of crowns	60	[Tanner et al., 2007]
	Compliance	2020 pN/nm	[Daniel et al., 1998]
Thin filament	Rest length	1119 nm	[Tanner et al., 2007]
	Binding-site spacing	12.4 nm	[Tanner et al., 2007]
	Rotation between sites	240°	[Tanner et al., 2007]
	Compliance	1743 pN/nm	[Daniel et al., 1998]

Chapter 5
CONCLUSIONS AND OPEN QUESTIONS

5.1 Conclusions

5.1.1 Two-spring cross-bridges are lattice spacing dependent

Single spring cross-bridges are insufficient to examine length-dependent properties of muscle, as they fail to incorporate the effects of changes in lattice spacing. Multi-spring models of the cross-bridge exist in both the axial and radial directions and are thus sensitive to changes in lattice spacing. Specifically, the force a multi-spring cross-bridge produces in both the axial and radial directions varies with lattice spacing. These multi-spring models offer new explanations of previously observed phenomena. Multi-spring models extend prior work, they do not contradict it. The axial force and kinetics of the multi-spring cross-bridges are similar to those of single spring models at neutral lattice spacings (those where cross-bridge energy is minimized) [Tanner et al., 2007, Daniel et al., 1998, Pate and Cooke, 1989]. The kinetics of the multi-spring models diverge from prior models when lattice spacing compresses or expands. Thus multi-spring models offer a way to determine the effects axial and radial forces which vary with lattice spacing, while being able to compare the results to prior work.

Two types of multi-spring models were compared in Chapter 2, a four-spring cross-bridge and a two-spring cross-bridge. A four-spring cross-bridge is able to draw parameters, such as spring rest length and angle, directly from experimentally measured values since each of its components is directly analogous to a section of the myosin head. The two-spring cross-bridge is a greater simplification of the cross-bridge and its lever arm, but produces substantially similar results, and does so in a fraction of the computational time required by the four-spring cross-bridge. As the two-spring cross-bridge is able to observe the same effects as a four-spring cross-bridge while incurring far lower computational costs, our spatially explicit models use two-spring cross-bridges as their mechanism of force generation.

5.1.2 Changes in lattice spacing increase the slope of the length-tension curve

The length-tension (LT) relationship has previously been attributed primarily to effects occurring in the axial direction [Rassier et al., 1999]. In Chapter 3 we showed that while change in the overlap of thick and thin filaments can account for 80% of the force change seen over the LT curve, changes in lattice spacing are necessary to produce the familiar steep LT curve. The LT curve which most closely fit literature values was that produced under isovolumetric conditions, where lattice spacing changes with sarcomere length to maintain a constant sarcomere volume. The isovolumetric LT curve combines the force reductions of sub-optimal lattice spacings with those of less than ideal overlaps. We confirmed the importance of lattice spacing to force production in skinned fiber experiments where sarcomere length was held constant while lattice spacing was osmotically varied as across the descending limb of the LT curve. The force produced by our skinned fiber bundles depended even more steeply on lattice spacing than in our models. This suggests that, if anything, our theoretical results underestimate the level of control lattice spacing exerts on force production.

Without these combined effects, we would have a shallower ascending limb which would result in smaller force changes for a given length transient. This reduced dependence of force on length would reduce the stability of oscillating systems that function on the ascending limb, such as cardiac muscle (as described by the Frank-Starling relationship) [Smith et al., 2009]. This suggests that we should rethink our classic model of the LT curve, substituting thinking which includes varying degrees of overlap and lattice spacings for that which considers only changes in overlap.

5.1.3 Cross-bridges store more energy than filament backbones in isometric contraction

Energy storage in muscular systems is necessary, both to reduce the cost of locomotion and to permit explosive movements. Unlike movement driving locomotion, explosive movement is defined as occurring non-cyclically. This non-cyclical movement is preceded by a period of energy build up where energy is integrated over time and sequestered as elastic strain [Lappin et al., 2006, Claverie et al., 2011]. This allows the power expended over the brief explosive

movement to be greater than that which muscle can sustain. A few studies have suggested that energy may be stored in muscle, either in deformation of either titin or of the thick and thin filaments [Monroy et al., 2007, Tidball and Daniel, 1986]. We found that the energy stored in deformation of the cross-bridges was more than twice that stored in deformation of the thick and thin filaments, at all lattice spacings and sarcomere lengths. This hints that cross-bridge deformation may be storing energy for use in producing explosive movements, a mechanism which can help to account for energy expended in such movements which doesn't appear to be present in deformation of accessory structures [Zack et al., 2009].

Our desire to examine the radial forces muscle generates was one of the original motivations behind the development of the multi-spring cross-bridge model. Improper regulation of radial force is implicated in dystrophic diseases [Batchelor and Winder, 2006]. The multi-filament model produces radial and axial forces of the same order of magnitude. This is similar to the levels of radial forces measured experimentally [Maughan and Godt, 1981b, Matsubara et al., 1984, Cecchi et al., 1990, Brenner and Yu, 1991b, Xu et al., 1993, Nyland and Maughan, 2000]. In Chapter 4 we showed that the level of radial force which is produced correlates very well with the energy stored in the cross-bridges, far better than does the axial force. This suggests that radial strain of the cross-bridges may be the primary driver of energy storage in the cross-bridges, and thus a previously unexamined means by which muscle stores energy.

5.2 Open Questions

5.2.1 Direct experimental measurement of radial force

Existing measurements of radial force are made indirectly, by examining lattice deformation during force generation or by deriving radial force from the radial stiffness of the contractile lattice as measured by atomic force microscopy. These measurements have several back-calculation steps that rely on estimated values and necessitate lumping the force needed to move fluid through the contractile lattice with the force resisting spacing changes. Some of the earliest observations of radial force suggest a more direct method [Hiramoto, 1956]. An injected droplet of immiscible oil in skinned or intact fibers deforms asymmetrically

on force generation, elongating into an oblate spheroid (as in Figure 5.1). The shape of this droplet can be quantified by relatively low magnification light microscopy. Techniques used in the study of compressed emulsions offer the ability to directly calculate the ratio of radial and axial compression necessary to produce the measured deformation [Mason et al., 1995, Lacasse et al., 1996]. This simple technique has the potential to offer rapid and direct measurements of the levels of radial force produced in intact and skinned muscle preparations, measurements which would greatly clarify the levels and regulation of radial forces. Use of this technique in dystrophic model systems could clarify whether radial forces are being improperly regulated at the sarcomeric level, or only at larger structural scales.

5.2.2 Modeling work

While the models we have built offer the world's most advanced look at spatial regulation, they were written to allow future modification and examination of other processes. The encapsulated model design (shown in the code listed in the Appendices) makes it relatively simple to introduce time dependent properties, new forms of regulation, or other forms of forces to future simulations.

Time dependent contractions

The simulations we performed were under isometric conditions; we did not vary sarcomere length. Allowing sarcomere length to change over the course of a single simulation would allow us to observe how compliant realignment and its effects on cross-bridge kinetics varies when the binding sites' locations are changing due to a method other than stretching [Daniel et al., 1998, Tanner et al., 2007]. Complaint realignment is the movement of a binding site relative to a myosin head due to force generated by and stretching of the filaments involved. While it would be trivially easy to change sarcomere length in a prescribed pattern, shortening against a simulated load would be a better replicant of conditions *in vivo*. A simple naturalistic load could be simulated by placing a movement-resisting damper element at the end of the sarcomere. Time varying sarcomere length would allow comparison to another class of contractions, permitting another form of model validation, and would

permit further examination of complaint realignment mechanisms.

Viscosity in the lattice

Perhaps the most exciting extension of the current model would be the inclusion of viscous shearing forces between the filaments as they shorten. All existing models treat the thick and thin filaments as though they were surrounded by a frictionless medium. Viscous coupling between filaments could be included in the model by attaching damping elements with constants determined by the relative movement of surrounding filaments. While the precise level of coupling between adjacent filaments is difficult to estimate due to the system's proximity to the boundary where continuum fluid dynamics breaks down, best estimates place it at a level that would effect unloaded shortening velocity [Daniel and Williams, 2012]. This would cause a reinterpretation of a classic measure of cross-bridge dynamics.

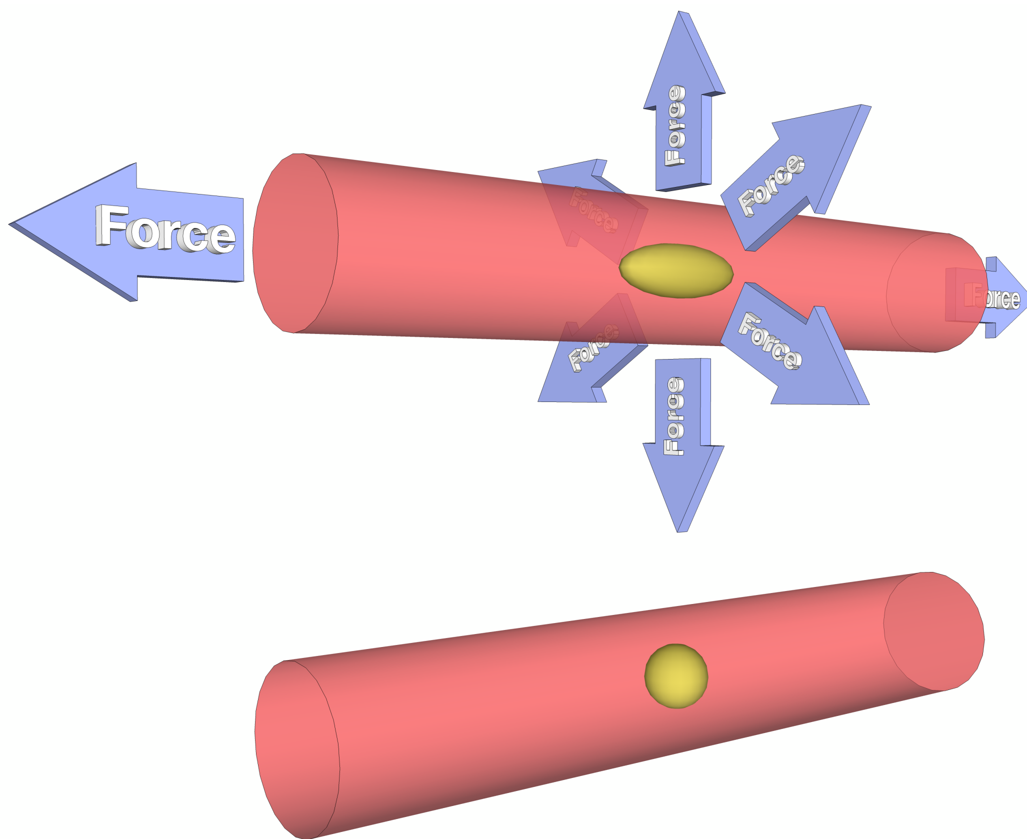


Figure 5.1: **Differences in axial and radial forces produces asymmetrical deformation of oil droplets in contracting fibers.** A droplet within a fiber elongates and becomes an oblate spheroid (from lower to upper image) when radial forces place it under compression.

BIBLIOGRAPHY

Bishow B Adhikari, Michael Regnier, Anthony J Rivera, Kareen L Kreutziger, and Donald A Martyn. Cardiac length dependence of force and force redevelopment kinetics with altered cross-bridge cycling. *Biophys J*, 87(3):1784–94, Sep 2004. doi: 10.1529/biophysj.103.039131.

Hind A Al-Khayat, Edward P Morris, Robert W Kensler, and John M Squire. Myosin filament 3D structure in mammalian cardiac muscle. *J Struct Biol*, 163(2):117–26, Aug 2008. doi: 10.1016/j.jsb.2008.03.011.

R McNeill Alexander. *Elastic mechanisms in animal movement.* Cambridge University Press, 1988.

R McNeill Alexander. Tendon elasticity and muscle function. *Comp Biochem Physiol, Part A Mol Integr Physiol*, 133(4):1001–11, Dec 2002. URL `http://www.sciencedirect.com/science/article/pii/S1095643302001435`.

R McNeill Alexander. Mechanics of animal movement. *Curr Biol*, 15(16):R616–9, Aug 2005. doi: 10.1016/j.cub.2005.08.016.

R McNeill Alexander and H C Bennet-Clark. Storage of elastic strain energy in muscle and other tissues. *Nature*, 265(5590):114–7, Jan 1977. doi: 10.1038/265114a0.

M Angela Bagni, Giovanni Cecchi, P J Griffiths, Y Maeda, G Rapp, and Christopher C Ashley. Lattice spacing changes accompanying isometric tension development in intact single muscle fibers. *Biophys J*, 67(5):1965–1975, Nov 1994. doi: 10.1016/S0006-3495(94)80679-4.

Clare L Batchelor and Steve J Winder. Sparks, signals and shock absorbers: how dystrophin loss causes muscular dystrophy. *Trends Cell Biol*, 16(4):198–205, Apr 2006. doi: 10.1016/j.tcb.2006.02.001.

Howard C Berg. *Random Walks in Biology.* Princeton University Press, 1993.

Derek J Blake, Andrew Weir, Sarah E Newey, and Kay E Davies. Function and genetics of dystrophin and dystrophin-related proteins in muscle. *Physiol Rev*, 82(2): 291–329, Apr 2002. doi: 10.1152/physrev.00028.2001.

M Blix. Die lange und die spannung des muskels. *Skand Arch Physiol*, 5:149–206, Mar 1894.

Bernhard Brenner. The stroke size of myosins: a reevaluation. *J Muscle Res Cell Motil*, 27(2):173–87, Jan 2006. doi: 10.1007/s10974-006-9056-7.

Bernhard Brenner and Leepo C Yu. Characterization of radial force and radial stiffness in Ca^{2+}-activated skinned fibres of the rabbit psoas muscle. *J Physiol*, 441:703–18, Sep 1991a.

Bernhard Brenner and Leepo C Yu. Characterization of radial force and radial stiffness in Ca^{2+}-activated skinned fibres of the rabbit psoas muscle. *J Physiol*, 441:703–18, Sep 1991b. URL `http://jp.physoc.org/content/441/1/703.long`.

Kenneth S Campbell. Interactions between connected half-sarcomeres produce emergent mechanical behavior in a mathematical model of muscle. *PLoS Comput Biol*, 5 (11):e1000560, Nov 2009. doi: 10.1371/journal.pcbi.1000560.

Olivier Cazorla, Yiming Wu, Thomas C Irving, and Henk Granzier. Titin-based modulation of calcium sensitivity of active tension in mouse skinned cardiac myocytes. *Circ Res*, 88(10):1028–35, May 2001.

Giovanni Cecchi, M Angela Bagni, P J Griffiths, Christopher C Ashley, and Y Maeda. Detection of radial crossbridge force by lattice spacing changes in intact single muscle fibers. *Science*, 250(4986):1409–11, Dec 1990.

P Bryant Chase, J Michael Macpherson, and Thomas L Daniel. A spatially explicit nanomechanical model of the half-sarcomere: myofilament compliance affects Ca^{2+}-activation. *Annals of biomedical engineering*, 32(11):1559–68, Nov 2004.

Thomas Claverie, Elliot Chan, and Sheila N Patek. Modularity and scaling in fast movements: power amplification in mantis shrimp. *Evolution*, 65(2):443–61, Feb 2011. doi: 10.1111/j.1558-5646.2010.01133.x.

M Cobb. Timeline: exorcizing the animal spirits: Jan Swammerdam on nerve function. *Nat Rev Neurosci*, 3(5):395–400, May 2002. doi: 10.1038/nrn806.

Thomas L Daniel and C David Williams. Modeling many molecular motors mostly motivated by moth movement. In *SICB Annual Meeting; Charleston, SC*, 2012.

Thomas L Daniel, Alan C Trimble, and P Bryant Chase. Compliant realignment of binding sites in muscle: transient behavior and mechanical tuning. *Biophys J*, 74(4): 1611–21, Jan 1998.

Julien S Davis and Neal D Epstein. Mechanistic role of movement and strain sensitivity in muscle contraction. *Proc Natl Acad Sci USA*, 106(15):6140–5, Apr 2009. doi: 10.1073/pnas.0812487106.

Jurriaan H de Groot and Johan L Van Leeuwen. Evidence for an elastic projection mechanism in the chameleon tongue. *Proc R Soc B*, 271(1540):761–70, Apr 2004. doi: 10.1098/rspb.2003.2637.

W L Delano. *The PyMOL Molecular Graphics System*. DeLano Scientific, Palo Alto, CA, USA., 2008. URL `http://www.pymol.org`.

Michael H Dickinson, Claire T Farley, Robert J Full, M A Koehl, R Kram, and S Lehman. How animals move: an integrative view. *Science*, 288(5463):100–6, Apr 2000. URL `http://www.sciencemag.org/content/288/5463/100.long`.

Michael H Dickinson, Gerrie P Farman, Mark A Frye, Tanya Bekyarova, David Gore, David W Maughan, and Thomas C Irving. Molecular dynamics of cyclically contracting insect flight muscle in vivo. *Nature*, 433(7023):330–4, Jan 2005. doi: 10.1038/nature03230.

Ken A Dill and Sarina Bromberg. *Molecular Driving Forces: Statistical Thermodynamics in Chemistry & Biology*. Garland Science, 2002.

Ken A Dill and Sarina Bromberg. *Molecular Driving Forces*. Garland Science, 2003.

K A P Edman. The velocity of unloaded shortening and its relation to sarcomere length and isometric force in vertebrate muscle fibres. *J Physiol*, 291:143–159, Jun 1979.

Edward H Egelman. The structure of F-actin. *J Muscle Res Cell Motil*, 6(2):129–51, Apr 1985.

C Ellington. The aerodynamics of hovering insect flight. VI. Lift and power requirements. *Phil Trans R Soc Lond B*, 305(1122):145–181, Feb 1984. doi: 10.1098/rstb.1984.0054.

R Fischetti, S Stepanov, G Rosenbaum, R Barrea, E Black, David Gore, R Heurich, E Kondrashkina, A J Kropf, S Wang, Ke Zhang, Thomas C Irving, and G B Bunker. The BioCAT undulator beamline 18ID: a facility for biological non-crystalline diffraction and x-ray absorption spectroscopy at the Advanced Photon Source. *J Synchrotron Radiat*, 11(Pt 5):399–405, Sep 2004. doi: 10.1107/S0909049504016760.

Franklin Fuchs and Donald A Martyn. Length-dependent Ca^{2+} activation in cardiac muscle: some remaining questions. *J Muscle Res Cell Motil*, 26(4-5):199–212, Jan 2005. doi: 10.1007/s10974-005-9011-z.

Franklin Fuchs and Yi-Peng Wang. Sarcomere length versus interfilament spacing as determinants of cardiac myofilament Ca^{2+} sensitivity and Ca^{2+} binding. *J Mol Cell Cardiol*, 28(7):1375–83, Jul 1996. doi: 10.1006/jmcc.1996.0129.

Nicole T George, C David Williams, H-M Hsu, Thomas C Irving, and Thomas L Daniel. The cross-bridge spring: cool muscles store elastic energy. *In Preparation*, 2012.

Robert E Godt and David W Maughan. Influence of osmotic compression on calcium activation and tension in skinned muscle fibers of the rabbit. *Pflugers Arch*, 391(4): 334–7, Oct 1981.

Y E Goldman. Measurement of sarcomere shortening in skinned fibers from frog muscle by white light diffraction. *Biophys J*, 52(1):57–68, 1987. doi: 10.1016/S0006-3495(87)83188-0.

Albert M Gordon, Andrew F Huxley, and Fred J Julian. The variation in isometric tension with sarcomere length in vertebrate muscle fibres. *J Physiol*, 184(1):170–92, Apr 1966. URL http://jp.physoc.org/content/184/1/170.abstract.

Albert M Gordon, Earl Homsher, and Michael Regnier. Regulation of contraction in striated muscle. *Physiol Rev*, 80(2):853–924, Apr 2000.

John Gosline, Margo Lillie, Emily Carrington, Paul Guerette, Christine Ortlepp, and Ken Savage. Elastic proteins: biological roles and mechanical properties. *Phil Trans R Soc Lond B*, 357(1418):121–32, Feb 2002. doi: 10.1098/rstb.2001.1022.

S Gourinath, Daniel M Himmel, Jerry H Brown, Ludmilla Reshetnikova, Andrew G Szent-Gyorgyi, and Carolyn Cohen. Crystal structure of scallop myosin S1 in the pre-power stroke state to 2.6 åresolution: flexibility and function in the head. *Structure*, 11(12):1621–7, Dec 2003.

W Gronenberg. The trap-jaw mechanism in the dacetine ants Daceton armigerum and Strumigenys sp. *J Exp Biol*, 199(Pt 9):2021–33, Jan 1996. URL http://jeb.biologists.org/content/199/9/2021.long.

Hideo Higuchi, T Yanagida, and Yale E Goldman. Compliance of thin filaments in skinned fibers of rabbit skeletal muscle. *Biophys J*, 69(3):1000–10, 1995. doi: 10.1016/S0006-3495(95)79975-1.

AV Hill. The heat of shortening and the dynamic constants of muscle. *Proc R Soc B*, 126(843):136–195, 1938.

Yukio Hiramoto. Physical state of muscle protoplasm. *Annotationes zoologicae Japonenses*, 29(2):63–69, Jun 1956. URL http://ci.nii.ac.jp/naid/110003353732.

Anne Houdusse and H Lee Sweeney. Myosin motors: missing structures and hidden springs. *Curr Opin Struct Biol*, 11(2):182–94, Apr 2001.

Anne Houdusse, Andrew G Szent-Gyorgyi, and Carolyn Cohen. Three conformational states of scallop myosin S1. *Proc Natl Acad Sci USA*, 97(21):11238–43, Oct 2000.

Jonathon Howard. *Mechanics of Motor Proteins and the Cytoskeleton*. Sinauer Associates, 2001.

A F Huxley. Muscle structure and theories of contraction. *Progress in biophysics and biophysical chemistry*, 7:255–318, Jan 1957.

A F Huxley. Mechanics and models of the myosin motor. *Philos Trans R Soc Lond, B, Biol Sci*, 355(1396):433–40, Jan 2000.

Andrew F Huxley and R Niedergerke. Structural changes in muscle during contraction; interference microscopy of living muscle fibres. *Nature*, 173(4412):971–3, May 1954. URL `http://www.nature.com/nature/journal/v173/n4412/abs/173971a0.html`.

Hugh E Huxley. The mechanism of muscular contraction. *Science*, 164(886):1356–65, Jun 1969.

Hugh E Huxley and Jean Hanson. Changes in the cross-striations of muscle during contraction and stretch and their structural interpretation. *Nature*, 173(4412):973–6, May 1954. URL `http://www.nature.com/nature/journal/v173/n4412/abs/173973a0.html`.

Thomas C Irving. X-ray diffraction of indirect flight muscle from Drosophila in vivo. *Nature's Versatile Engine: Insect Flight Muscle Inside and Out. JO Vigoreaux, editor. Landes Bioscience, Georgetown, TX*, pages 197–213, 2006.

Thomas C Irving and David W Maughan. In vivo x-ray diffraction of indirect flight muscle from Drosophila melanogaster. *Biophys J*, 78(5):2511–5, May 2000. doi: 10.1016/S0006-3495(00)76796-8.

Thomas C Irving, John P Konhilas, D Perry, R Fischetti, and Pieter P de Tombe. Myofilament lattice spacing as a function of sarcomere length in isolated rat myocardium. *Am J Physiol Heart Circ Physiol*, 279(5):H2568–73, Nov 2000. URL `http://ajpheart.physiology.org/content/279/5/H2568.long`.

Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: Open source scientific tools for Python, 2001. URL `http://www.scipy.org/`.

William M Kier and Johan L Van Leeuwen. A kinematic analysis of tentacle extension in the squid Loligo pealei. *J Exp Biol*, 200(Pt 1):41–53, Jan 1997. URL `http://jeb.biologists.org/content/200/1/41.long`.

Jan Köhler, Gerhard Winkler, Imke Schulte, Tim Scholz, William McKenna, Bernhard Brenner, and Theresia Kraft. Mutation of the myosin converter domain alters cross-bridge elasticity. *Proc Natl Acad Sci USA*, 99(6):3557–62, Mar 2002. doi: 10.1073/pnas.062415899.

Kareen L Kreutziger, Todd E Gillis, Jonathan P Davis, Svetlana B Tikunova, and Michael Regnier. Influence of enhanced troponin-C Ca^{2+}-binding affinity on cooperative thin filament activation in rabbit skeletal muscle. *J Physiol (Lond)*, 583(Pt 1): 337–50, Aug 2007. doi: 10.1113/jphysiol.2007.135426.

M Lacasse, G Grest, D Levine, T Mason, and D Weitz. Model for the elasticity of compressed emulsions. *Phys Rev Lett*, 76(18):3448–3451, Apr 1996. doi: 10.1103/PhysRevLett.76.3448.

A Kristopher Lappin, Jenna A Monroy, Jason Q Pilarski, Eric D Zepnewski, David J Pierotti, and Kiisa C Nishikawa. Storage and recovery of elastic potential energy powers ballistic prey capture in toads. *J Exp Biol*, 209(Pt 13):2535–53, Jul 2006. doi: 10.1242/jeb.02276.

Jun Liu, Shenping Wu, Mary C Reedy, Hanspeter Winkler, Carmen Lucaveche, Yifan Cheng, Michael K Reedy, and Kenneth A Taylor. Electron tomography of swollen rigor fibers of insect flight muscle reveals a short and variably angled s2 domain. *J Mol Biol*, 362(4):844–60, Sep 2006. doi: 10.1016/j.jmb.2006.07.084.

Donald A Martyn, Bishow B Adhikari, Michael Regnier, Jin Gu, Sengen Xu, and Leepo C Yu. Response of equatorial x-ray reflections and stiffness to altered sarcomere length and myofilament lattice spacing in relaxed skinned cardiac muscle. *Biophys J*, 86(2):1002–11, Feb 2004. doi: 10.1016/S0006-3495(04)74175-2.

T Mason, J Bibette, and D Weitz. Elasticity of compressed emulsions. *Phys Rev Lett*, 75(10):2051–2054, Sep 1995. doi: 10.1103/PhysRevLett.75.2051.

Ichiro Matsubara, Yale E Goldman, and Robert Simmons. Changes in the lateral filament spacing of skinned muscle fibres when cross-bridges attach. *J Mol Biol*, 173 (1):15–33, 1984. doi: 10.1016/0022-2836(84)90401-7.

David W Maughan and Robert E Godt. Radial forces within muscle fibers in rigor. *J Gen Physiol*, 77(1):49–64, Jan 1981a. doi: 10.1085/jgp.77.1.49.

David W Maughan and Robert E Godt. Radial forces within muscle fibers in rigor. *Journal of General Physiology*, 77(1):49–64, Jan 1981b. doi: 10.1085/jgp.77.1.49.

J M Metzger and R L Moss. Shortening velocity in skinned single muscle fibers. influence of filament lattice spacing. *Biophys J*, 52(1):127–131, 1987. doi: 1010.1016/S0006-3495(87)83197-1.

Barry M Millman. The filament lattice of striated muscle. *Physiol Rev*, 78(2):359–91, Apr 1998. URL `http://physrev.physiology.org/cgi/content/full/78/2/359`.

Jenna A Monroy, A Kristopher Lappin, and Kiisa C Nishikawa. Elastic properties of active muscle–on the rebound? *Exerc Sport Sci Rev*, 35(4):174–9, Oct 2007. doi: 10.1097/jes.0b013e318156e0e6.

L R Nyland and David W Maughan. Morphology and transverse stiffness of Drosophila myofibrils measured by atomic force microscopy. *Biophys J*, 78(3):1490–7, Mar 2000. doi: 10.1016/S0006-3495(00)76702-6.

C Pasternak, S Wong, and E L Elson. Mechanical function of dystrophin in muscle cells. *J Cell Biol*, 128(3):355–61, Feb 1995. URL `http://jcb.rupress.org/content/128/3/355.long`.

Edward Pate and Roger Cooke. A model of crossbridge action: the effects of ATP, ADP and P_i. *J Muscle Res Cell Motil*, 10(3):181–96, Jun 1989.

S N Patek, D M Dudek, and M V Rosario. From bouncy legs to poisoned arrows: elastic movements in invertebrates. *J Exp Biol*, 214(Pt 12):1973–80, 2011.

Erwin J G Peterman, Hernando Sosa, and W E Moerner. Single-molecule fluorescence spectroscopy and microscopy of biomolecular motors. *Annu Rev Phys Chem*, 55:79–96, Jan 2004. doi: 10.1146/annurev.physchem.55.091602.094340.

G Rapp, Christopher C Ashley, M Angela Bagni, Peter J Griffiths, and Giovanni Cecchi. Volume changes of the myosin lattice resulting from repetitive stimulation of single muscle fibers. *Biophys J*, 75(6):2984–95, Dec 1998. doi: 10.1016/S0006-3495(98)77739-2.

D E Rassier, B R MacIntosh, and W Herzog. Length dependence of active force production in skeletal muscle. *J Appl Physiol*, 86(5):1445–57, May 1999. URL `http://jap.physiology.org/content/86/5/1445.long`.

Ivan Rayment, Wojciech Rypniewski, Karen Schmidt-Bäse, Robert Smith, Diana Tomchick, Matthew M Benning, Donald Winkelmann, Gary Wesenberg, and Hazel Holden. Three-dimensional structure of myosin subfragment-1: A molecular motor. *Science*, 261(5117):50–58, Jul 1993. URL `http://www.jstor.org/stable/2881462`.

M K Reedy, K C Holmes, and R T Tregear. Induced changes in orientation of the cross-bridges of glycerinated insect flight muscle. *Nature*, 207(5003):1276–80, Sep 1965.

Mary C Reedy. Visualizing myosin's power stroke in muscle contraction. *J Cell Sci*, 113:3551–62, Oct 2000.

Thomas J Roberts and Emanuel Azizi. Flexible mechanisms: the diverse roles of biological springs in vertebrate movement. *J Exp Biol*, 214(Pt 3):353–61, Feb 2011. doi: 10.1242/jeb.038588.

Mark Schoenberg. Geometrical factors influencing muscle force development. I. The effect of filament spacing upon axial forces. *Biophys J*, 30(1):51–67, Jan 1980a.

Mark Schoenberg. Geometrical factors influencing muscle force development. II. Radial forces. *Biophys J*, 30(1):69–77, Jan 1980b.

B Seebohm, F Matinmehr, Jan Köhler, A Francino, F Navarro-Lopéz, A Perrot, C Özcelik, William McKenna, B Brenner, and Theresia Kraft. Cardiomyopathy mutations reveal variable region of myosin converter as major element of cross-bridge compliance. *Biophys J*, 97(3):806–24, Aug 2009. doi: 10.1016/j.bpj.2009.05.023.

D Smith, Michael A Geeves, John Sleep, and S M Mijailovich. Towards a unified theory of muscle contraction. I: Foundations. *Annals of biomedical engineering*, 36 (10):1624–40, Jan 2008.

L Smith, C Tainter, Michael Regnier, and Donald A Martyn. Cooperative cross-bridge activation of thin filaments contributes to the frank-starling mechanism in cardiac muscle. *Biophys J*, 96(9):3692–702, May 2009. doi: 10.1016/j.bpj.2009.02.018.

James A Spudich. The myosin swinging cross-bridge model. *Nature Reviews Molecular Cell Biology*, 2(5):387–92, May 2001. doi: 10.1038/35073086.

John M Squire. X-ray diffraction methods in muscle research. In *The Structural Basis of Muscular Contraction*. Plenum Press, 1983.

S Suzuki and H Sugi. Extensibility of the myofilaments in vertebrate skeletal muscle as revealed by stretching rigor muscle fibers. *J Gen Physiol*, 81(4):531–46, Apr 1983.

Y Takagi, H Shuman, and Yale E Goldman. Coupling between phosphate release and force generation in muscle actomyosin. *Philos Trans R Soc Lond, B, Biol Sci*, 359 (1452):1913–20, Dec 2004. doi: 10.1098/rstb.2004.1561.

Bertrand C W Tanner, Thomas L Daniel, and Michael Regnier. Sarcomere lattice geometry influences cooperative myosin binding in muscle. *PLoS Comput Biol*, 3(7): e115, Jul 2007. doi: 10.1371/journal.pcbi.0030115.

Kenneth A Taylor, H Schmitz, Mary C Reedy, Yale E Goldman, Clara Franzini-Armstrong, Hiroyuki Sasaki, Richard T Tregear, K Poole, Carmen Lucaveche, R J Edwards, Li Fan Chen, Hanspeter Winkler, and Michael K Reedy. Tomographic 3D reconstruction of quick-frozen, Ca^{2+}-activated contracting insect flight muscle. *Cell*, 99(4):421–31, Nov 1999.

J G Tidball and Thomas L Daniel. Elastic energy storage in rigored skeletal muscle cells under physiological loading conditions. *Am J Physiol*, 250(1 Pt 2):R56–64, Jan 1986. URL `http://ajpregu.physiology.org/content/250/1/R56.reprint`.

Francisco Torrent-Guasp, Mladen J Kocica, Antonio F Corno, Masashi Komeda, Francesc Carreras-Costa, A Flotats, Juan Cosin-Aguillar, and Han Wen. Towards new understanding of the heart structure and function. *Eur J Cardiothorac Surg*, 27 (2):191–201, Feb 2005. doi: 10.1016/j.ejcts.2004.11.026.

Michael S Tu and Thomas L Daniel. Submaximal power output from the dorsolongitudinal flight muscles of the hawkmoth Manduca sexta. *J Exp Biol*, 207(Pt 26): 4651–62, Dec 2004. doi: 10.1242/jeb.01321.

Taro Q P Uyeda, Paul D Abramson, and James A Spudich. The neck region of the myosin motor domain acts as a lever arm to generate movement. *Proc Natl Acad Sci USA*, 93(9):4459–64, Apr 1996.

C David Williams, Michael Regnier, and Thomas L Daniel. Axial and radial forces of cross-bridges depend on lattice spacing. *PLoS Comput Biol*, 6(12):e1001018, Jan 2010. doi: 10.1371/journal.pcbi.1001018.

Sengen Xu, Bernhard Brenner, and Leepo C Yu. State-dependent radial elasticity of attached cross-bridges in single skinned fibres of rabbit psoas muscle. *J Physiol*, 465: 749–65, Jun 1993. URL `http://jp.physoc.org/content/465/1/749.long`.

T I Zack, Thomas Claverie, and Sheila N Patek. Elastic energy storage in the mantis shrimp's fast predatory strike. *J Exp Biol*, 212(Pt 24):4002–9, Dec 2009. doi: 10.1242/jeb.034801.

Appendix A

APPENDIX A

This appendix contains the Python code which generates a single cross-bridge with four, two, or a single spring. It is a companion to Chapter 2 and generated the data used therein.

A.1 File crossbridge.py

```
#!/usr/bin/env python
# encoding: utf-8
""" Crossbridge.py
Created by Dave Williams on 2009-07-01.

This file defines the system for the various crossbridge types.
These crossbridges have four points that may be represented as
linear or torsional springs. A schematic follows::
           H   - Head of the myosin
           |
           G   - Globular domain
           |
           C   - Converter region
          /
         N     - Neck region
        /
    ===T====== - Thick filament
"""

import warnings
import numpy.random as random
from scipy.optimize import fmin_bfgs as fmin
from numpy import pi, sqrt, log
import math as m


class Spring(object):
    """A generic spring that handles some accounting"""
    def __init__(self, spring_config):
        """Create the spring with a set of passed values

        Takes:
```

```
33              spring_config: a dictionary containing the keys
34                  weak: the spring rest value when unbound or weakly ↩
                        bound
35                  strong: the spring rest value when strongly bound
36                  spring_konstant: the spring constant
37          Returns:
38              None
39          """
40          self.weak = spring_config['weak']
41          self.strong = spring_config['strong']
42          self.k = spring_config['spring_konstant']
43          # Diffusion related
44          temperature = 288 # in K
45          boltzman = 1.381 * 10**-23 # Boltzman const (in J/K)
46          k_t = boltzman * temperature * 10**21  # kT without pN/nM ↩
                conversion
47          # Normalize is a factor used to normalize the PDF of the ↩
                segment vals
48          self.normalize = sqrt(2*pi*k_t/self.k)
49          self.stand_dev = sqrt(k_t/self.k) # of seg vals
50
51      def rest(self, state):
52          """Return the rest value of the spring at in a given state
53
54          Takes:
55              state: the spring state of interest [1, 2, or 3]
56          Returns:
57              None
58          """
59          if state in [1, 2]:
60              return self.weak
61          elif state == 3:
62              return self.strong
63          else:
64              warnings.warn("Improper value for spring state")
65
66      def energy(self, curr_val, state):
67          """Given a current value, return the energy the spring ↩
                stores"""
68          if state in [1, 2]:
69              return (0.5 * self.k * m.pow((curr_val-self.weak), 2))
70          elif state == 3:
71              return (0.5 * self.k * m.pow((curr_val-self.strong), 2))
72          else:
73              warnings.warn("Improper value for spring state")
74
75      def bop(self):
76          """Bop the xb to a new location, based on an exponential ↩
                distribution
```

```
77          of energies for each independent segment of the ↩
               crossbridge as
78          determined by Boltzmann's law. Return the new head location.
79
80          Justification of technique
81          --------------------------
82          We are dealing with the energy stored in a spring, this ↩
               has a
83          dependence on the displacement of the spring thusly:
84              U(x) = 1/2 k x^2
85          Where k is the spring constant of that spring and x is ↩
               really x-xs,
86          where xs is the rest length of the spring. At x=xs the ↩
               spring sees no
87          strain and U(x)=0. If we are looking for the probability ↩
               that the
88          particle or myosin head on the end of the spring will be ↩
               in a given
89          location at a given time, we can fall back onto ↩
               Boltzmann's law which
90          tells us that:
91              p(x)/p(xs) = exp(-U(x)/kT)/exp(-U(xs)/kT)
92          Where p in this instance is the relative probability of ↩
               finding the
93          particle at x (as opposed to xs), k is Boltzmann's ↩
               constant, and T is
94          the system's temperature. This could also be represented as
95          P(x)/P(xs). Where P(x) is the absolute probability of ↩
               being located at
96          position x. We would like to deal with the absolute ↩
               probability and so
97          must find Z, a factor that allows us to normalize the ↩
               distribution
98          such that
99              \int_{-\inf}^\inf 1/Z exp(-U(x)/(kT)) dx = 1
100         This is calculated in Mathematica by taking the inverse of ↩
               the
101         integral
102             \int_{-\inf}^\inf 1/Z \exp(-.5 k x^2 /(kT)) dx
103         This evaluates to
104             Z = \sqrt{2 pi kT / k}
105         Meaning we are presented with a probability density ↩
               function of
106             P(x) = \sqrt{k / (2 pi kT)} \exp(-(k x^2)/(2 kT))
107         This looks very similar to the PDF of a normal distribution,
108         typically written as:
109             NormDis(x) = 1/sqrt(2 pi sig^2) exp(-(x-mu)^2/(2 sig^2))
110         Where sig is the standard deviation of the distribution ↩
               and mu is its
```

```
111         mean. This means that we can interpert our desired PDF as ↩
               that of a
112         normal distribution having:
113             mu = xs
114             sig = sqrt(kT / k)
115
116         Details of implementation
117         -------------------------
118         This makes use of numpy's normal distribution:
119             numpy.random.normal(loc=0.0, scale=1.0, size=None)
120         Where loc is the mean is mu and scale is the standard ↩
               deviation is sig
121         such that the probability density function of the resulting
122         distribution is:
123             f(x) = 1/sqrt(2 pi sig^2) exp(-(x-mu)^2/(2 sig^2))
124         If we feed numpy's exponential distribution mu=xs and ↩
               sig=sqrt(kT/k),
125         we have a distribution of:
126             f(x) = 1/sqrt(s pi kT / k) exp(-(x-xs)^2/(2 ↩
                   sqrt(kT/k)^2))
127         Or, simplifiying:
128             f(x) = sqrt(k / (2 pi kT)) exp(-k (x-xs)^2/(2 kT))
129         The PDF that we derived above (with an explicit xs term ↩
               this time).
130         We can customize this for each spring with which we deal ↩
               with by
131         plugging in their own means and spring constants.
132         """
133         return (random.normal(self.weak, self.stand_dev))
134
135
136 class Crossbridge(object):
137     def __init__(self, config = None):
138         """A generic cross-bridge, a TNCG one by default
139
140         This cross-bridge class provides default functionality, such
141         as calculation of energies and transition rate constants, ↩
               that
142         later classes of cross-bridge may build upon.
143         Takes:
144             config: a dictionary containing the spring ↩
                   definitions, a sample:
145                 {'T': {'weak': 40*pi/180,
146                        'strong': 40*pi/180,
147                        'spring_konstant': 100
148                       },
149                  'N': ...
150                  'C': ...
151                  'G': ...
152                 }
```

```
153         """
154         # Eventually, take out default config, put in GenerateData
155         if config == None:
156             config = {
157                 'T': {
158                     'weak': 40*pi/180,
159                     'strong': 40*pi/180,
160                     'spring_konstant': 100
161                 },
162                 'N': {
163                     'weak':  10.5,
164                     'strong':  10.5,
165                     'spring_konstant': 10
166                 },
167                 'C': {
168                     'weak':2*pi - 165*pi/180,
169                     'strong': 2*pi - 110*pi/180,
170                     'spring_konstant': 40
171                 },
172                 'G': {
173                     'weak': 9.6,
174                     'strong': 9.6,
175                     'spring_konstant': 5
176                 }
177             }
178         self.config = config
179         # Define initial spring values
180         self.t = Spring(self.config['T'])
181         self.n = Spring(self.config['N'])
182         self.c = Spring(self.config['C'])
183         self.g = Spring(self.config['G'])
184 
185     def minimize_energy(self, h_loc, state):
186         """The cross-bridge's minimum energy for a given head ↩
               location, state
187 
188         Takes:
189             h_loc:  [x,y] location of the cross-bridge tip
190             state: state of the cross-bridge, 1, 2, or 3
191         Returns:
192             energy: cross-bridge's minimum energy
193             min_conv: [x,y] converter location yielding the ↩
                   minimum energy
194         """
195         rest_conv_loc = (self.n.rest(state) * ↩
               m.cos(self.t.rest(state)),
196                          self.n.rest(state) * ↩
                                m.sin(self.t.rest(state)))
197         min_conv = fmin(self.energy,
198             rest_conv_loc, args = (h_loc, state), disp=0)
```

```
199             return (self.energy(min_conv, h_loc, state), min_conv)
200
201         def energy(self, conv_loc, h_loc, state):
202             """Return the energy in the xb with the given parameters"""
203             (t_ang, n_len, c_ang, g_len) = self.seg_values(conv_loc, ↩
                    h_loc)
204             return float(
205                 self.t.energy(t_ang, state) +
206                 self.n.energy(n_len, state) +
207                 self.c.energy(c_ang, state) +
208                 self.g.energy(g_len, state)
209             )
210
211         def free_energy(self, h_loc, state):
212             """Return the free energy in the xb with the given ↩
                    parameters"""
213             g_0 = 13 #in RT
214             atp_conc = 0.005 # or 5 mM
215             adp_conc = 0.00003 # or 30 uM
216             phos_conc  = 0.003 # or 3 mM
217             g_lib = - g_0 - log(atp_conc / (adp_conc * phos_conc))
218             alph = 0.28 #G_lib freed in 0->1 trans, from ↩
                    Bert/Tom/Pate/Cooke
219             eta = 0.68 #ditto, for 1->2 trans
220             if state is 1:
221                 return float(0)
222             elif state is 2:
223                 return float(alph * g_lib + ↩
                        self.minimize_energy(h_loc, state)[0])
224             elif state is 3:
225                 return float(eta * g_lib + self.minimize_energy(h_loc, ↩
                        state)[0])
226
227         def force(self, h_loc, state):
228             """From the head loc, the force vector being exerted by ↩
                    the XB"""
229             (energy, conv_loc) = self.minimize_energy(h_loc, state)
230             (t_ang, n_len, c_ang, g_len) = self.seg_values(conv_loc, ↩
                    h_loc)
231             del(energy, t_ang, n_len) # Not needed
232             c_k = self.c.k
233             g_k = self.g.k
234             c_s = self.c.rest(state)
235             g_s = self.g.rest(state)
236             f_x = (-g_k * (g_len - g_s) * m.cos(c_ang) +
237                    1/g_len * c_k * (c_ang - c_s) * m.sin(c_ang))
238             f_y = (-g_k * (g_len - g_s) * m.sin(c_ang) +
239                    1/g_len * c_k * (c_ang - c_s) * m.cos(c_ang))
240             return [float(f_x), float(f_y)]
241
```

```
242     def seg_values(self, conv_loc, h_loc):
243         """Calculate the values of the segments of the XB"""
244         diff = [h_loc[0] - conv_loc[0], h_loc[1] - conv_loc[1]]
245         t_ang = m.atan2(conv_loc[1], conv_loc[0])
246         n_len = m.hypot(conv_loc[0], conv_loc[1])
247         c_ang = m.atan2(diff[1], diff[0]) + m.pi - t_ang
248         g_len = m.hypot(diff[0], diff[1])
249         return (t_ang, n_len, c_ang, g_len)
250
251     def bind_or_not(self, b_site):
252         """Given an (x,y) location of an open binding site, bind ↩
               or not after
253         bopping the cross-bridge head to a new location. Return a ↩
               boolean,
254         True for a binding event and False for no binding event.
255         """
256         ## Bop the springs to get new values
257         t_ang = self.t.bop()
258         n_len = self.n.bop()
259         c_ang = self.c.bop()
260         g_len = self.g.bop()
261         ## Translate those values to (x,y) postitions
262         conv_loc = (n_len * m.cos(t_ang),
263                     n_len * m.sin(t_ang))
264         h_loc = (conv_loc[0] + g_len * m.cos(c_ang + t_ang - m.pi),
265                  conv_loc[1] + g_len * m.sin(c_ang + t_ang - m.pi))
266         ## Find the distance to the binding site
267         distance = m.hypot(b_site[0]-h_loc[0], b_site[1]-h_loc[1])
268         ## The binding prob is dept on the exp of a dist
269         b_prob = 12*m.exp(-distance**2)
270         ## Throw a random number to check binding
271         return (b_prob > random.rand())
272
273     def r12(self, b_site, trials):
274         """Give the prob of binding, given a b_site and number of ↩
               trials """
275         # Binds gives us the number of times binding occurs out of ↩
               all trials
276         binds = sum(self.bind_or_not(b_site) for t in range(trials))
277         return (float(binds)/float(trials))
278
279     def r23(self, b_site):
280         """Given a binding site, b_site, to which a myosin head is ↩
               loosely
281         bound, return a probability of transition to a tightly ↩
               bound state
282         """
283         state2_energy = self.minimize_energy(b_site, 2)[0]
284         state3_energy = self.minimize_energy(b_site, 3)[0]
```

```
285         rate = .1 * (1 + m.tanh(.4 * (state2_energy - ↩
               state3_energy)+4))+.001
286         # Note that the .001 is just to keep rates above 0.0000 at ↩
               all times
287         return float(rate)
288
289     def r31(self, b_site):
290         """Given a binding site, b_site, to which a myosin head is ↩
               tightly
291         bound, return a probability of transition to an unbound ↩
               state
292         """
293         state3_energy = self.minimize_energy(b_site, 3)[0]
294         #rate = m.exp(-1 / (state3_energy + 1e-9)) #1e-9 avoids ↩
               1/0 at rest loc
295         rate =  m.sqrt(.01 * state3_energy) + 0.02
296         return float(rate)
297
298
299 class FourSpring(Crossbridge):
300     """An instance of the four-spring crossbridge.
301
302            H    - Head of the myosin
303            |
304            G    - Globular domain,   linear spring
305            |
306            C    - Converter region,  torsional spring
307           /
308          N      - Neck region,       linear spring
309         /
310     ===T====== - Thick filament,    torsional spring
311     """
312     def __init__(self, config = None):
313         """No modification of values needed, just trigger the att ↩
               calc"""
314         Crossbridge.__init__(self, config)
315
316
317 class TwoSpring(Crossbridge):
318     """An instance of the two-spring crossbridge.
319
320            H    - Head of the myosin
321            |
322            G    - Globular domain,   linear spring
323            |
324            C    - Converter region,  torsional spring
325           /
326          N      - Neck region,       fixed length
327         /
328     ===T====== - Thick filament,    fixed angle
```

```
329     """
330     def __init__(self, config = None):
331         """Modify values for this spring system, trigger the ↩
            attribute calc"""
332         Crossbridge.__init__(self, config)
333 
334     def minimize_energy(self, h_loc, state):
335         """Return the min energy in the XB with the head at the ↩
            given loc"""
336         rest_conv_loc = (self.n.rest(state) * ↩
            m.cos(self.t.rest(state)),
337                          self.n.rest(state) * ↩
                                m.sin(self.t.rest(state)))
338         return (self.energy(rest_conv_loc, h_loc, state), ↩
            rest_conv_loc)
339 
340     def bind_or_not(self, b_site):
341         """Given an (x,y) location of an open binding site, bind ↩
            or not after
342         bopping the cross-bridge head to a new location. Return a ↩
            boolean,
343         True for a binding event and False for no binding event.
344         """
345         ## Bop the springs to get new values
346         t_ang = self.t.rest(1)
347         n_len = self.n.rest(1)
348         c_ang = self.c.bop()
349         g_len = self.g.bop()
350         ## Translate those values to (x,y) postitions
351         conv_loc = (n_len * m.cos(t_ang),
352                     n_len * m.sin(t_ang))
353         h_loc = (conv_loc[0] + g_len * m.cos(c_ang + t_ang - m.pi),
354                  conv_loc[1] + g_len * m.sin(c_ang + t_ang - m.pi))
355         ## Find the distance to the binding site
356         distance = m.hypot(b_site[0]-h_loc[0], b_site[1]-h_loc[1])
357         ## The binding prob is dept on the exp of a dist
358         b_prob = 72*m.exp(-distance**2) +.00001
359         ## Throw a random number to check binding
360         return (b_prob > random.rand())
361 
362 
363 class OneSpring(Crossbridge):
364     """An instance of the one-spring crossbridge"""
365     def __init__(self, config = None):
366         Crossbridge.__init__(self, config)
367 
368     def minimize_energy(self, h_loc, state):
369         """Return the min energy of the XB at h_loc, ignore y ↩
            dimension"""
370         # Ignore y dim and only use energy in neck
```

```
371         return (self.n.energy(h_loc[0], state), (h_loc[0], 0))
372 
373     def bind_or_not(self, b_site):
374         """Given an (x,y) location of an open binding site, bind ↩
                or not after
375         bopping the cross-bridge head to a new location. Return a ↩
                boolean,
376         True for a binding event and False for no binding event.
377         """
378         ## Bop the spring to get a new value
379         n_len = self.n.bop()
380         ## Find the distance to the binding site
381         distance = abs(b_site[0]-n_len) #Ignore y dim
382         ## The binding prob is dept on the exp of a dist
383         b_prob = m.exp(-distance)
384         ## Throw a random number to check binding
385         return (b_prob > random.rand())
```

Appendix B

APPENDIX B

This appendix lists the python code which comprises the model of the half-sarcomere used in Chapters 3 and 4. This is split across several files.

B.1 File af.py

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3  """
4  af.py - An actin filament
5
6  Create and maintain a thin filament and the subgroups that ↩
      comprise it.
7
8  Created by Dave Williams on 2010-01-04.
9  """
10
11 import numpy as np
12
13
14 class BindingSite(object):
15     """A singular globular actin site"""
16     def __init__(self, parent_thin_fil, thin_index, orientation):
17         """Create a binding site on the thin filament
18
19         Parameters:
20             parent_thin_fil: the calling thin filament instance
21             thin_index: the axial index on the parent thin filament
22             orientation: select between six orientations (0-5)
23         """
24         # Remember passed attributes
25         self.parent_thin = parent_thin_fil
26         self.thin_index = thin_index
27         # Use the passed orientation index to choose the correct
28         # orientation vector according to schema in ThinFilament ↩
              docstring
29         orientation_vectors = ((0.866, -0.5), (0, -1), (-0.866, ↩
              -0.5),
```

```
30                  (-0.866, 0.5), (0, 1), (0.866, 0.5))
31          self.orientation = orientation_vectors[orientation]
32          # Create attributes to store things not yet present
33          self.bound_to = None # None if unbound, Crossbridge object ↩
                otherwise
34          self.thick_face = None
35
36      def __repr__(self):
37          """Return the current situation of the binding site"""
38          ident = ['Binding Site #' + str(self.thin_index) + ' Info']
39          ident.append(14 * '=')
40          ident.append('State: ' + str(self.get_state()))
41          if self.get_state() != 0:
42              ident.append('Forces: ' + str(self.axialforce())
43                           + '/' + str(self.radialforce()))
44          return '\n'.join(ident)
45
46      def axialforce(self, axial_location=None):
47          """Return the axial force of the bound cross-bridge, if any
48
49          Parameters:
50              axial_location: location of the current node (optional)
51          Returns:
52              f_x: the axial force generated by the cross-bridge
53          """
54          if self.bound_to is None:
55              return 0.0
56          # Axial force on actin is equal but opposite
57          return -self.bound_to.axialforce(tip_axial_loc = ↩
                axial_location)
58
59      def radialforce(self):
60          """Radial force vector of the bound cross-bridge, if any
61
62          Returns:
63              (f_y, f_z): the radial force vector of this binding site
64          """
65          if self.bound_to is None:
66              return 0.0
67          force_mag = -self.bound_to.radialforce() # Equal but ↩
                opposite
68          return np.multiply(force_mag, self.orientation)
69
70      def link_thick_face(self, thick_face):
71          """Create link to the relevant thick filament face when ↩
                known"""
72          self.thick_face = thick_face
73
74      def bind_to(self, crossbridge):
75          """Link this binding site to a cross-bridge object"""
```

```
76          self.bound_to = crossbridge
77
78      def unbind(self):
79          """Kill off any link to a crossbridge"""
80          assert(self.bound_to is not None) # Else why try to unbind?
81          self.bound_to = None
82
83      def get_state(self):
84          """Return the current numerical state, 0/unbound or ↩
               1/bound"""
85          return self.bound_to is not None
86
87      def get_lattice_spacing(self):
88          """Get lattice spacing from the parent filament"""
89          return self.parent_thin.get_lattice_spacing()
90
91      def get_axial_location(self):
92          """Return the current axial location of the binding site"""
93          return self.parent_thin.axial[self.thin_index]
94
95
96  class ThinFace(object):
97      """Represent one face of an actin filament
98      Deals with orientation in the typical fashion for thin filaments
99          ================
100         ||     m4     ||  ^
101         || m3      m5 ||  |    ^
102         ||     af     ||  Z   /
103         || m2      m0 ||     X
104         ||     m1     ||       Y-->
105         ================
106     """
107     def __init__(self, thin_fil, orientation, binding_sites):
108         """Create the thin filament face
109
110         Parameters:
111             thin_fil: the thin filament on which this face sits
112             orientation: which myosin face is opposite this face ↩
                    (0-5)
113             binding_sites: links to the actin binding sites on ↩
                    this face
114         """
115         self.parent_filament = thin_fil
116         self.orientation = orientation
117         self.binding_sites = binding_sites
118         self.thick_face = None  # ThickFace instance this face ↩
               interacts with
119
120     def nearest(self, axial_location):
121         """Where is the nearest binding site?
```

```
122
123         There a fair number of cases that must be dealt with here. ↩
               When
124         the system becomes too short (and some nearest queries are ↩
               being
125         directed to a thin face that doesn't really have anything ↩
               near
126         that location) the face will just return the nearest ↩
               location and
127         let the kinetics deal with the fact that binding is about ↩
               as likely
128         as stepping into the same river twice.
129
130         Parameters:
131             axial_location: the axial coordinates to seek a match ↩
                   for
132         Return:
133             binding_site: the nearest binding site on this face
134         """
135         # Next two lines of code enforce a jittery hiding, ↩
               sometimes the
136         # binding site just beyond the hiding line can be accessed
137         hiding_line = self.parent_filament.get_hiding_line()
138         axial_location = max(hiding_line, axial_location)
139         face_locs = [site.get_axial_location() for site in ↩
               self.binding_sites]
140         close_index = np.searchsorted(face_locs, axial_location)
141         # If not using a very short SL, where the end face loc is ↩
               closest
142         if close_index != len(face_locs):
143             dists = np.abs((face_locs[close_index] - axial_location,
144                             face_locs[close_index-1] - ↩
                                   axial_location))
145         else:
146             return self.binding_sites[close_index-1] # If so, ↩
                   return end
147         if dists[0] < dists[1] or len(self.binding_sites) >= ↩
               close_index + 1:
148             return self.binding_sites[close_index]
149         else:
150             return self.binding_sites[close_index + 1]
151
152     def radialforce(self):
153         """What is the radial force this face experiences?
154
155         A side note: This was where the attempt to write the model ↩
               out in
156         a functional manner broke down. I got this far with ↩
               nothing ever
```

```
157         asking another instance for any information and everything ↩
               being
158         passed by method parameters. This was a really nice idea and
159         worked well until this point where I had to start performing
160         overly complex mental calisthenics to understand how ↩
               things were
161         going to be passed around. This lead to the current system ↩
               where
162         each instance has an internal state that it is responsible ↩
               for
163         keeping. This might make debugging harder in the long run, ↩
               but it
164         made the model writable in the meanwhile. Some teeth ↩
               gnashing is
165         included below for reference.
166
167         Teeth gnashing:
168             The source of conflict here seems to be a competition ↩
                   between
169             the desire to write this in a functional manner and ↩
                   have all
170             information passed down to the function as is needed ↩
                   and the
171             desire to be able to call any function of any module ↩
                   at any
172             time and have it return something sensible. This makes
173             testing some bits easier but means that it can become ↩
                   harder
174             to track what is going on with the states of the various
175             functions. I am unsure as to how this should be ↩
                   resolved at
176             this time. I want the final design to be as ↩
                   uncluttered and
177             easy to troubleshoot as is possible. Perhaps something ↩
                   where
178             the storage of information is kept separate from the ↩
                   ways that
179             the modules are acting upon it? The advantage of this ↩
                   is that
180             passing information around becomes infinitely easier, ↩
                   the
181             drawback is that I am not sure that this isn't just a ↩
                   step
182             removed from declaring every variable to be global and ↩
                   making
183             the whole thing a fair bit more brittle.
184
185         Returns:
186             radial_force: the radial force myosin heads on this ↩
                   face exert
```

```
187         """
188         # First, a sanity check
189         if self.thick_face is None:
190             raise AttributeError("Thick filament not assigned yet.")
191         # Now find the forces on each cross-bridge
192         radial_forces = [site.radialforce() for site in ↩
                self.binding_sites]
193         return np.sum(radial_forces, 1)
194 
195     def set_thick_face(self, myosin_face):
196         """Link to the relevant myosin filament."""
197         assert(self.orientation == myosin_face.orientation)
198         self.thick_face = myosin_face
199         return
200 
201     def axial_location(self, binding_site_id):
202         """Get the axial location of the selected binding site"""
203         return ↩
                self.binding_sites[binding_site_id].get_axial_location()
204 
205     def get_lattice_spacing(self):
206         """Return lattice spacing to the face's opposite number"""
207         return self.parent_filament.get_lattice_spacing()
208 
209 
210 class ThinFilament(object):
211     """Each thin filament is made up of two actin strands.  The ↩
            overall
212     filament length at rest is 1119 nm [(1)][Tanner2007].  Each ↩
            strand
213     hosts 45 actin binding sites (or nodes) giving the whole ↩
            filament
214     90 actin nodes, plus one at the Z-line for accounting.
215 
216     These nodes are located every 24.8 nm on each actin strand and ↩
            are
217     rotated by 120 degrees relative to the prior node ↩
            [(1)][Tanner2007].
218     This organization does not specify the relative offsets of the ↩
            two
219     filament's nodes.
220 
221     ## Naive repeating geometry of the thin filament
222     The binding nodes of the two actin filaments must be offset by a
223     multiple of the angle (120 degrees)x(distance offset/(24.8 ↩
            nm)), but
224     not by 360 degrees, or one of the actin filaments would have ↩
            no binding
225     sites facing a neighboring thick filament.  We assume that the ↩
            actin
```

```
226     nodes on the two strands are offset by half of the distance ↩
           between
227     adjacent nodes (12.4 nm) and 180 degrees.  This means that if ↩
           one actin
228     filament has a binding site facing one myosin filament, the ↩
           second actin
229     filament will have a binding site facing a second myosin ↩
           filament
230     12.4 nm down the thin filament.  The second myosin filament ↩
           will be
231     240 degrees clockwise of the first myosin filament around the
232     thin filament.
233
234     ## Binding site numbering
235     As in the thick filament, the nodes/binding sites on the thin ↩
           filament
236     are numbered from low at the left to high on the right. Thus ↩
           the 90th
237     node is adjacent to the Z-line.
238     """
239     def __init__(self, parent_lattice, face_orientations, ↩
           z_line=1250,
240                  start=0):
241         """Initialize the thin filament
242
243         Parameters:
244             parent_lattice: the calling half-sarcomere instance
245             face_orientations: list of faces' numerical ↩
                   orientation (0-5)
246             z_line: the location of the end of the thin filament ↩
                   (1250 nm)
247             start: which of the 26 actin monomers in an actin
248                 repeating unit this filament begins with (defaults
249                 to the first)
250         Returns:
251             None
252         ## Thin face arrangement
253         The thin filaments faces correspond to the following ↩
               diagram:
254         ================
255         ||     m4     ||  ^
256         || m3      m5 ||  |    ^
257         ||     af     ||  Z   /
258         || m2      m0 ||     X
259         ||     m1     ||       Y-->
260         ================
261         These orientations correspond to the orientations of the ↩
               facing
262         thick filaments. Each thin filament will link to either ↩
               faces
```

```
263         0, 2, and 4 or faces 1, 3, and 5.
264         This will result in a set of unit vectors pointing from the
265         thin filament to the thick faces that are either
266         ((0, 1), (0.866, -0.5), (-0.866, -0.5))
267         for the case on the left or, for the case on the right,
268         ((-0.886, 0.5), (0.866, 0.5), (0, -1))
269         The vectors govern both what radial force linking ↩
                cross-bridges
270         generate and which actin monomers are considered to be ↩
                facing
271         the adjacent filaments.
272         """
273         # Remember who created you
274         self.parent_lattice = parent_lattice
275         # TODO The creation of the monomer positions and angles ↩
                should be refactored into a static function of similar.
276         # Figure out axial positions
277         mono_per_poly = 26 # actin monomers in an actin polymer unit
278         poly_per_fil = 15 # actin polymers in a thin filament
279         polymer_base_length = 72.0 # nm per polymer unit length
280         polymer_base_turns = 12.0 # revolutions per polymer
281         rev = 2*np.pi # one revolution
282         pitch = polymer_base_turns * rev / mono_per_poly
283         rise = polymer_base_length / mono_per_poly
284         # Monomer positions start near the m-line
285         monomer_positions = [(z_line - ↩
                mono_per_poly*poly_per_fil*rise) + m*rise
286                 for m in range(mono_per_poly*poly_per_fil)]
287         monomer_angles = [(((m+start+1) % mono_per_poly) * pitch) ↩
                % rev
288                 for m in range(mono_per_poly * poly_per_fil)]
289         # Convert face orientations to angles, then to angles from ↩
                0 to 2pi
290         orientation_vectors = ((0.866, -0.5), (0, -1.0), (-0.866, ↩
                -0.5),
291                 (-0.866, 0.5), (0, 1.0), (0.866, 0.5))
292         face_vectors = [orientation_vectors[o] for o in ↩
                face_orientations]
293         face_angles = [np.arctan2(v[1], v[0]) for v in face_vectors]
294         face_angles = [v + rev if (v < 0) else v for v in ↩
                face_angles]
295         # Find which monomers are opposite each face
296         # TODO Convert this to use a distance rather than an ↩
                angle...
297         wiggle = rev/24 # count faces within 15 degrees of opposite
298         mono_in_each_face = ↩
                [np.nonzero(np.abs(np.subtract(monomer_angles,
299                  face_angles[i]))<wiggle)[0] for i in ↩
                    range(len(face_angles))]
```

```
300         # This is [(index_to_face_1, ...), (index_to_face_2, ...), ↩
                ...]
301         # Translate monomer position to binding site position
302         axial_by_face = [[monomer_positions[mono_ind] for mono_ind ↩
                in face]
303                 for face in mono_in_each_face]
304         axial_flat = np.sort(np.hstack(axial_by_face))
305         # Tie the nodes on each face into the flat axial locations
306         node_index_by_face = np.array([[np.nonzero(axial_flat == ↩
                l)[0][0]
307             for l in f] for f in axial_by_face])
308         face_index_by_node = np.tile(None, len(axial_flat))
309         for face_ind in range(len(node_index_by_face)):
310             for node_ind in node_index_by_face[face_ind]:
311                 face_index_by_node[node_ind] = face_ind
312         # Create binding sites and thin faces
313         self.binding_sites = []
314         for index in range(len(axial_flat)):
315             orientation = ↩
                    face_orientations[face_index_by_node[index]]
316             self.binding_sites.append(BindingSite(self, index, ↩
                    orientation))
317         self.thin_faces = []
318         for face_index in range(len(node_index_by_face)):
319             face_binding_sites = map(lambda i: ↩
                    self.binding_sites[i],
320                     node_index_by_face[face_index])
321             orientation = face_orientations[face_index]
322             self.thin_faces.append(
323                     ThinFace(self, orientation, face_binding_sites))
324         del(orientation, face_binding_sites)
325         # Remember the axial locations, both current and rest
326         self.axial = axial_flat
327         self.rests = np.diff(np.hstack([self.axial, z_line]))
328         # Other thin filament properties to remember
329         self.number_of_nodes = len(self.binding_sites)
330         self.thick_faces = None # Set after creation of thick ↩
                filaments
331         self.z_line = z_line
332         self.k = 1743
333
334     def set_thick_faces(self, thick_faces):
335         """Set the adjacent thick faces and associated values
336
337         Parameters:
338             thick_faces: links to three surrounding thick faces, ↩
                    in the
339                 order (0, 2, 4) or (1, 3, 5)
340
341         ## Myosin filament arrangement
```

```
342         ==================================   ^
343         ||       m4       ||   m3       m5   ||   |    ^
344         ||                or       af       ||   Z   /
345         ||       af       ||                ||     X
346         ||   m2       m0   ||       m1       ||       Y-->
347         ==================================
348         """
349         self.thick_faces = thick_faces
350         for a_face, m_face in zip(self.thin_faces, ↩
                self.thick_faces):
351             a_face.set_thick_face(m_face)
352 
353     def effective_axial_force(self):
354         """The axial force experienced at the Z-line from the thin ↩
                filament
355 
356         This only accounts for the force at the Z-line due to the ↩
                actin
357         node adjacent to it, i.e. this is the force that the Z-line
358         experiences, not the tension existing elsewhere along the ↩
                thin
359         filament.
360         Return:
361             force: the axial force at the Z-line
362         """
363         return (self.rests[-1] - (self.z_line - self.axial[-1])) * ↩
                self.k
364 
365     def axial_force_of_each_node(self, axial_locations=None):
366         """Return a list of the thin filament axial force at each ↩
                node
367 
368         Parameters:
369             axial_locations: location of each node (optional)
370         Returns:
371             axial_forces: a list of the axial force at each node
372         """
373         if axial_locations == None:
374             axial_forces = [site.axialforce() for site in ↩
                    self.binding_sites]
375         else:
376             axial_forces = [site.axialforce(loc) for
377                     site,loc in zip(self.binding_sites, ↩
                            axial_locations)]
378         return axial_forces
379 
380     def axialforce(self, axial_locations=None):
381         """Return a list of axial forces at each binding site node ↩
                location
382 
```

```
383         This returns the force at each node location (including ↩
               the z-disk
384         connection point), this is the sum of the force that ↩
               results from
385         displacement of the nodes from their rest separation and ↩
               the axial
386         force created by any bound cross-bridges
387
388         Parameters:
389             axial_locations: location of each node (optional)
390         Return:
391             force: sum of force from the cross-bridges and node ↩
                   displacement
392         """
393         # Calculate the force exerted by the thin filament's ↩
               backbone
394         thin = self._axial_thin_filament_forces(axial_locations)
395         # Calculate the force exerted by any existing cross-bridges
396         binding_sites = ↩
               self.axial_force_of_each_node(axial_locations)
397         # Return the combination of the two
398         return np.add(thin, binding_sites)
399
400     def radial_force_of_each_node(self):
401         """The radial force produced at each binding site node
402
403         Parameters:
404             None
405         Returns
406             radial_forces: a list of (f_y, f_z) force vectors
407         """
408         radial_forces = [nd.radialforce() for nd in ↩
               self.binding_sites]
409         return radial_forces
410
411     def radial_force_of_filament(self):
412         """The sum of the radial force experienced by this filament
413
414         Parameters:
415             None
416         Returns:
417             radial_force: a single (f_y, f_z) vector
418         """
419         radial_force_list = self.radial_force_of_each_node()
420         radial_force = np.sum(radial_force_list, 0)
421         return radial_force
422
423     def stress(self):
424         """A metric of how much the thin filament locations are ↩
               offset.
```

```
425
426             The quality of this metric is unproven.
427
428             Parameters:
429                 None
430             Returns:
431                 stress: the sum of the thin filament nodes' ↩
                        displacements
432             """
433             dists = np.diff(np.hstack([self.axial, self.z_line]))
434             return np.sum(np.abs(dists - self.rests))
435
436         def _axial_thin_filament_forces(self, axial_locations=None):
437             """The force of the filament binding sites, sans ↩
                    cross-bridges
438
439             Parameters:
440                 axial_locations: location of each node (optional)
441             Returns:
442                 net_force_on_each_binding_site: per-site force
443             """
444             # Use the thin filament's stored axial locations if none ↩
                    are passed
445             if axial_locations == None:
446                 axial_locations = np.hstack([self.axial, self.z_line])
447             else:
448                 axial_locations = np.hstack([axial_locations, ↩
                        self.z_line])
449             # Find the distance from binding site to binding site
450             dists = np.diff(axial_locations)
451             # Find the compressive or expansive force on each spring
452             spring_force = (dists - self.rests) * self.k
453             # The first node's not connected, so that side has no ↩
                    force...
454             spring_force = np.hstack([0, spring_force])
455             # Convert this to the force on each node
456             net_force_on_each_binding_site = np.diff(spring_force)
457             return net_force_on_each_binding_site
458
459         def update_axial_locations(self, flat_axial_locs):
460             """Update the axial locations to the passed ones
461
462             Parameters:
463                 flat_axial_locs: the new locations for all axial nodes
464             Returns:
465                 None
466             """
467             # You aren't allowed to change the number of nodes
468             assert(len(flat_axial_locs) == len(self.axial))
469             self.axial = flat_axial_locs
```

```
470
471     def get_hiding_line(self):
472         """Return the distance below which actin binding sites are ↩
               hidden"""
473         return self.parent_lattice.get_hiding_line()
474
475     def get_binding_site(self, index):
476         """Return a link to the binding site site at index"""
477         return self.binding_sites[index]
478
479     def get_bound_sites(self):
480         """Give a list of binding sites that are bound to an XB"""
481         return filter(lambda bs: bs.bound_to is not None, ↩
               self.binding_sites)
482
483     def get_axial_location(self, index):
484         """Return the axial location of the node at index"""
485         return self.axial[index]
486
487     def get_lattice_spacing(self):
488         """Return the lattice spacing of the half-sarcomere"""
489         return self.parent_lattice.get_lattice_spacing()
490
491     @property
492     def face(self, face_index): # FIXME Current point, trying to ↩
            get property accessing of the thin_faces down, but I am ↩
            not sure how indexing gets passed to a property, is it ↩
            like a normal attribute [2] or like a function call (2)?
493         """Return the thin face of the passed index"""
494         return self.thin_faces[face_index]
495
496
497 if __name__ == '__main__':
498     print("af.py is really meant to be called as a supporting ↩
            module")
```

B.2 File mf.py

```
 1 #!/usr/bin/env python
 2 # encoding: utf-8
 3 """
 4 mf.py - A myosin thick filament
 5
 6 Created by Dave Williams on 2010-01-04.
 7 """
 8
 9 import mh
10 import numpy as np
11
```

```
12
13 class Crown(object):
14     """Three cross-bridges on a thick filament at a given axial ↩
          location
15
16     Crowns are a physiologically relevant division of the thick ↩
          filament.
17     They are clusters of three cross-bridges that hang off of the ↩
          thick
18     filament, separated from each other by a 120 degree rotation ↩
          about the
19     thick filament's pitch. In this model, they serve as force ↩
          distribution
20     nodes; any axial or radial force that a crown's cross-bridge ↩
          generates
21     is felt equally by the other two cross-bridges.
22     """
23     def __init__(self, parent_thick, thick_index,
24             cross_bridges, orientations):
25         """Create the myosin crown
26
27         Parameters:
28             parent_thick: the calling thick filament instance
29             thick_index: the axial index on the parent thick ↩
                  filament
30             cross_bridges: the cross-bridges that populate the crown
31             orientations: select between two crown orientations (0 ↩
                  or 1)
32         """
33         # Remember the passed attributes
34         self.parent_thick = parent_thick
35         self.thick_index = thick_index
36         self.crossbridges = cross_bridges
37         # Use the passed orientation (type 0 or 1) to create the ↩
              orientation
38         # vectors that the crown uses to pass back proper radial ↩
              forces
39         # NB: vectors are ((face_0,face_2, face_3), (face_1, ↩
              face_3, face_5))
40         crown_vectors = (((-0.886, 0.5), (0.866, 0.5), (0, -1)),
41                 ((0, 1), (0.866, -0.5), (-0.866, -0.5)))
42         self.orientations = crown_vectors[orientations]
43
44     def axialforce(self, axial_location=None):
45         """The sum of the axial force generated by each head"""
46         axial_force = [xb.axialforce(axial_location) for
47                 xb in self.crossbridges]
48         return sum(axial_force)
49
50     def radialforce(self):
```

```
51          """Radial the sum of force generated by each head as a ↩
               vector (y,z)"""
52          radial_forces = []
53          for crossbridge, orient in zip(self.crossbridges, ↩
               self.orientations):
54              force_mag = crossbridge.radialforce()
55              radial_forces.append(np.multiply(force_mag, orient))
56          return np.sum(radial_forces, 0)
57
58      def transition(self):
59          """Put each head through a transition cycle """
60          transitions = [xb.transition() for xb in self.crossbridges]
61          return transitions
62
63      def get_lattice_spacing(self):
64          """Do what it says on the tin"""
65          return self.parent_thick.get_lattice_spacing()
66
67      def _set_timestep(self, timestep):
68          """Set the length of time step used to calculate ↩
               transitions"""
69          [xb._set_timestep(timestep) for xb in self.crossbridges]
70
71      def get_axial_location(self):
72          """Do what it says on the tin, return this crown's axial ↩
               location"""
73          return ↩
               self.parent_thick.get_axial_location(self.thick_index)
74
75
76  class ThickFace(object):
77      """Represent one face of a thick filament
78
79      Thick filaments have six faces, arranged thus:
80       ---------------------
81       | Myosin face order  |
82       |--------------------|
83       |         a1         |
84       |     a0       a2    |   ^
85       |         mf         |   |
86       |     a5       a3    |   Z
87       |         a4         |    Y-->
88       ---------------------
89      Because myosin crowns are arranged in the following pattern, ↩
           faces will
90      either have one cross-bridge ever 42.9nm or a repeating ↩
           pattern wherein
91      each set has a cross-bridge, a gap of 28.6 nm, a cross-bridge, ↩
           and a
92      final gap of 14.3 nm before the pattern repeats.
```

```
93      -----------------------------------------------------
94      | Crown Level 1   | Crown Level 2   | Crown Level 3   |
95      |  0nm offset     | 14.3 nm later   | 14.3 nm later   |
96      |-----------------|-----------------|-----------------|
97      |       a1        |       a1        |       a1        |
98      |    a0     a2    |    a0 |  a2     |    a0     a2    |
99      |      \M1/      -->     /M1\      -->     \M1/       |
100     |    a5 |  a3     |    a5     a3    |    a5 |  a3     |
101     |       a4        |       a4        |       a4        |
102     -----------------------------------------------------
103     Further discussion is located in the "ThickFilament" ↩
           documentation.
104     """
105     def __init__(self, parent_filament, axial_locations, thin_face,
106             orientation, start):
107         """Instantiate the thick filament face with its heads
108 
109         Parameters:
110             parent_filament: the thick filament supporting this face
111             axial_locations: the axial locations of nodes along ↩
                   the face,
112                  we want this list kept linked to the filament's ↩
                       version
113             thin_face: the thin filament face located opposite
114             orientation: the numerical orientation of this face ↩
                   (0-5)
115             start: what crown level this face starts on (1, 2, or 3)
116         """
117         # Remember the calling parameters
118         self.parent_filament = parent_filament
119         self.axial_locations = axial_locations
120         self.thin_face = thin_face
121         self.orientation = orientation # numerical orientation (0-5)
122         # Instantiate the cross-bridges along the face
123         self.xb = []
124         self.xb_by_crown = [] # Includes levels with no heads
125         crown_level = start
126         # For faces in positions 0, 2, or 4 ...
127         if orientation in (0, 2, 4):
128             # look at each thick filament crown location ...
129             for i in range(len(axial_locations)):
130                 # and add cross-bridges at the appropriate ↩
                       locations.
131                 if crown_level in (1, 3):
132                     head = mh.Crossbridge(i, self, thin_face)
133                     self.xb.append(head)
134                     self.xb_by_crown.append(head)
135                 elif crown_level == 2:
136                     self.xb_by_crown.append(None)
```

```
137                 # Increment the crown level, cycling back to 1 ↩
                       after 3.
138                 crown_level = crown_level % 3 + 1
139         elif orientation in (1, 3, 5):
140             for i in range(len(axial_locations)):
141                 if crown_level in (1, 3):
142                     self.xb_by_crown.append(None)
143                 elif crown_level == 2:
144                     head = mh.Crossbridge(i, self, thin_face)
145                     self.xb.append(head)
146                     self.xb_by_crown.append(head)
147                 crown_level = crown_level % 3 + 1
148         # Record the thick filament node index at which ↩
               cross-bridge sits
149         self.xb_index = [xb.face_index for xb in self.xb]
150 
151     def __repr__(self):
152         """The string representation of the thick filament face
153         The representation is as follows:
154             Thick -    |====================
155             Bindings -   |  |   |  |  \ \
156             Bindings         | |  | |   \   \
157             Thin -       ------------------------------|
158             Where | is a loosely bound XB and \ is strongly bound
159         """
160         thick = '|' + len(self.xb) * '='
161         xb_string = [' ', '|', '\\']
162         thickbnd = ' ' + ''.join([xb_string[xb.get_numeric_state()]
163                                   for xb in self.xb])
164         thinbnd = 12*' ' + ''.join([xb_string[act.get_state()]
165                                   for act in ↩
                                       self.thin_face.binding_sites])
166         thin = 12*' ' + len(self.thin_face.binding_sites) * '-' + ↩
               '|'
167         return (thick + '\n' + thickbnd + '\n' + thinbnd + '\n' + ↩
               thin + '\n')
168 
169     def axialforce(self):
170         """Return the total axial force of the face's ↩
               cross-bridges"""
171         axial = [crossbridge.axialforce() for crossbridge in ↩
               self.xb]
172         return sum(axial)
173 
174     def radialtension(self):
175         """Sum of the absolute values of radial force for every ↩
               myosin"""
176         radial = [crossbridge.radialforce() for crossbridge in ↩
               self.xb]
177         return sum(radial)
```

```
178
179     def radialforce(self):
180         """The radial force this face experiences
181
182         Parameters:
183             None:
184         Returns:
185             radial_force: sum of radial force of each myosin along ↩
                    the face
186         """
187         radial = [crossbridge.radialforce() for crossbridge in ↩
                self.xb]
188         return sum(radial)
189
190     def transition(self):
191         """Give each of the face's cross-bridges a chance to ↩
                transition"""
192         transitions = [crossbridge.transition() for crossbridge in ↩
                self.xb]
193         return transitions
194
195     def get_xb(self, crossbridge_index = None):
196         """Return an XB of interest or a list of all the face's ↩
                XBs"""
197         if crossbridge_index is None:
198             return self.xb
199         else:
200             return self.xb_by_crown[crossbridge_index]
201
202     def get_axial_location(self, crossbridge_index):
203         """Return the axial location of a cross-bridge"""
204         return self.parent_filament.axial[crossbridge_index]
205
206     def get_states(self):
207         """Return the numeric states (0,1,2) of all cross-bridges"""
208         return [xb.get_numeric_state() for xb in self.xb]
209
210     def get_lattice_spacing(self):
211         """Return lattice spacing to the face's opposite number"""
212         return self.parent_filament.get_lattice_spacing()
213
214
215 class ThickFilament(object):
216     """The thick filament is a string of myosin crowns
217
218     It is attached to the m-line at one end and to nothing
219     at the other (yet).
220     """
221     def __init__(self, parent_lattice, thin_faces, start):
222         """Initialize the thick filament.
```

```
223
224         Parameters:
225             parent_lattice: the calling half-sarcomere instance
226             thin_faces: links to six surrounding actin filament ↩
                    faces
227             start: initial crown level (1-3)
228
229         ## Actin filament arrangement
230         The actin filament list should be as follows:
231
232                 a1          ^
233             a0      a2      |   ^
234                 mf          z  /
235             a5      a3        x
236                 a4             y-->
237
238         ## Crown orientations
239         The rotation of neighboring crowns is different than that ↩
                used in
240         existing spatially explicit models, and is taken from new ↩
                analysis
241         of mammalian cardiac muscle.  Thick filaments sprout ↩
                myosin crowns
242         every 14.3nm, with a repeating pattern of azimuthal ↩
                perturbation
243         (rotation around the thick filament's long axis) every ↩
                three crown
244         lengths [(1)][AlKhayat2008].  The azimuthal perturbation ↩
                is such
245         that the first and third crowns in any 43nm repeat are ↩
                rotated by
246         60 degrees from the second crown's orientation ↩
                [(1)][AlKhayat2008].
247
248         This relates to the nearby actin filaments such that the ↩
                crowns
249         come in two configurations, linked to either actin ↩
                filaments 0, 2,
250         and 4 or 1, 3, and 5. They are arranged in an "A, B, A, A, ↩
                B, A,
251         ..." repeating pattern.
252
253         ## Crown spacing and thick filament length
254         Each half thick filament is 858 nm long and consists of 60 ↩
                myosin
255         nodes and one node at the M-line [(2)][Tanner2007].  As ↩
                each of the
256         myosin nodes is the location of a 3-myosin crown, each ↩
                half-thick
```

```
257         filament will have 180 myosins, slightly more than the 150 ↩
               present
258         in mammalian striated muscle [(2)][Tanner2007].  The ↩
               M-line side of
259         the thick filament has an initial bare zone of from 80 nm
260         [(3)][Higuchi1995] to 58 nm [(2)][Tanner2007].
261
262         The crowns are on a 43 nm repeat, with three crowns per ↩
               repeat.
263         This means that each crown will be spaced 43/3 = 14.3 nm ↩
               apart.
264
265         ## Orientation and parsing into faces
266
267         The thick filament is also organized into faces, collections
268         of cross-bridges that are opposite opposing acting faces. ↩
               These
269         six faces provide another way to organize the thick ↩
               filament,
270         one from which it is more easy to group all the interactions
271         that occur between the thick filament and one of its ↩
               adjacent
272         thin filaments. The main drawback of these faces is that ↩
               they
273         are subject to irregular cross-bridge distributions as a ↩
               result
274         of what "crown level" the thick filament starts on and as ↩
               a result
275         of the fact that three of the adjacent thin filaments get ↩
               more
276         opportunities to interact with the thick filament than do ↩
               the
277         other three thin filaments (see the documentation of the ↩
               thick
278         filament faces for more information). The cross-bridge ↩
               patterns
279         of the thick filament faces with various initial conditions
280         are shown below.
281
282         ### Opposite actin faces 0, 2, and 4
283         ||============================================================||
284         ||      Start at crown level 1 ↩
                                            ||
285         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
286         || Line |    \       \   \       \   \       \     <--XBs ↩
                 ||
287         ||============================================================||
288         ||      Start at crown level 2 ↩
                                            ||
```

```
289         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
290         || Line |        \   \       \   \       \   \     <--XBs ↩
                  ||
291         ||============================================================||
292         ||      Start at crown level 3 ↩
                                         ||
293         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
294         || Line |    \   \       \   \       \   \         <--XBs ↩
                  ||
295         ||============================================================||
296
297         ### Opposite actin faces 1, 3, and 5
298         ||============================================================||
299         ||      Start at crown level 1 ↩
                                         ||
300         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
301         || Line |        \           \           \         <--XBs ↩
                  ||
302         ||============================================================||
303         ||      Start at crown level 2 ↩
                                         ||
304         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
305         || Line |    \           \           \             <--XBs ↩
                  ||
306         ||============================================================||
307         ||      Start at crown level 3 ↩
                                         ||
308         || M    |----|---|---|---|---|---|---|---|---|-... ↩
               <--Nodes ||
309         || Line |            \           \           \     <--XBs ↩
                  ||
310         ||============================================================||
311
312         ## Other parameters
313         The spring constant of the thick filament, the sections ↩
               connecting
314         myosin crowns, is 2020 pN/nm [(4)][Daniel1998].
315
316         [AlKhayat2008]:http://dx.doi.org/10.1016/j.jsb.2008.03.011
317         [Tanner2007]:http://dx.doi.org/10.1371/journal.pcbi.0030115
318         [Higuchi1995]:http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1236329
319         [Daniel1998]:http://dx.doi.org/10.1016/S0006-3495(98)77875-0
320         """
321         # Remember who created you
322         self.parent_lattice = parent_lattice
323         # Create a list of crown axial locations and relevant data
```

```
324         bare_zone = 58 # Length of the area before any crowns
325         crown_spacing = 14.3 # Spacing between adjacent crowns
326         n_cr = 60 # Number of myosin crowns
327         self.axial = [bare_zone + n*crown_spacing for n in ↩
                range(n_cr)]
328         self.rests = np.diff(np.hstack([0, self.axial]))
329         # Instantiate the faces
330         self.thick_faces = []
331         for face_index in range(len(thin_faces)):
332             self.thick_faces.append(ThickFace(self, self.axial,
333                 thin_faces[face_index], face_index, start))
334         # Find the crown levels (1, 2, or 3) and orientation vectors
335         crown_levels = [(n+start-1)%3+1 for n in range(n_cr)]
336         crown_orientations = [0 + (l == 2) for l in crown_levels]
337         # Instantiate the myosin crowns
338         self.crowns = []
339         # For each crown position...
340         for index in range(n_cr):
341             crown_xbs = []
342             # look at the cross-bridges for each face...
343             for face in self.thick_faces:
344                 current_xb = face.get_xb(index)
345                 # and store it and its face if the cross-bridge ↩
                        exists.
346                 if current_xb is not None:
347                     crown_xbs.append(current_xb)
348             # Create a crown with these faces and cross-bridges.
349             self.crowns.append(Crown(self, index, crown_xbs,
350                 crown_orientations[index]))
351         # Thick filament properties to remember
352         self.number_of_crowns = n_cr
353         self.thin_faces = thin_faces
354         self.k = 2020 # Spring constant of thick filament in pN/nm
355         self.b_z = bare_zone
356 
357     def __repr__(self):
358         """String representation of the thick filament"""
359         faces = '' .join(["Face " + str(face.orientation) + ": \n" +
360                     face.__repr__() + '\n'
361                     for face in self.thick_faces])
362         return faces
363 
364     def effective_axial_force(self):
365         """Get the axial force generated at the M-line
366 
367         This looks only at the force due to the crown next to the
368         M-line, as this is the only point on the thick filament that
369         can /directly/ generate force upon the M-line. It does not
370         account for internal strain along the other nodes or force ↩
                due
```

```
371         to bound cross-bridges.
372         It is assumed that the M-line is at an x location of 0.
373         Return:
374             force: the axial force at the M-line
375         """
376         return (self.axial[0] - self.b_z) * self.k
377 
378     def axial_force_of_each_crown(self, axial_locations=None):
379         """Return a list of the axial force on each crown"""
380         if axial_locations == None:
381             axial_force = [cr.axialforce() for cr in self.crowns]
382         else:
383             axial_force = [cr.axialforce(loc) for
384                     cr,loc in zip(self.crowns, axial_locations)]
385         return axial_force
386 
387     def axialforce(self, axial_locations=None):
388         """Return a list of axial forces at each crown location
389 
390         This returns the force at each crown, accounting for the
391         internal strain of the thick filament and the force ↩
               generated
392         by bound cross-bridge heads.
393         Parameters:
394             axial_locations: location of each crown (optional)
395         Return:
396             force: the sum of the axial forces generated by all ↩
                    crowns
397         """
398         # Calculate the force exerted by the thick filament's ↩
               backbone
399         thick = self._axial_thick_filament_forces(axial_locations)
400         # Retrieve the force each crown generates
401         crown = self.axial_force_of_each_crown(axial_locations)
402         # Return the combination of backbone and crown forces
403         return np.add(thick, crown)
404 
405     def radialtension(self):
406         """The radial tension this filament experiences
407 
408         Parameters:
409             None
410         Returns:
411             radial_tension: the sum of the absolute value of the ↩
                    radial
412              force that each cross-bridge along the filament ↩
                     experiences
413         """
414         face_tensions = [face.radialtension() for face in ↩
               self.thick_faces]
```

```
415             return sum(face_tensions)
416
417         def radial_force_of_each_crown(self):
418             """Return a list of the radial force vectors (y,z) of each ↩
                    crown"""
419             radial_forces = [cr.radialforce() for cr in self.crowns]
420             return radial_forces
421
422         def radial_force_of_filament(self):
423             """Gives the radial force generate by the entire filament
424
425             Parameters:
426                 None
427             Return:
428                 radial_force: (y,z) vector of radial force from all ↩
                        crowns"""
429             # Retrieve the force all crowns generate
430             crown_forces = self.radial_force_of_each_crown()
431             # Return the combination of all crown forces
432             return np.sum(crown_forces, 0)
433
434         def stress(self):
435             """A metric for how offset the nodes are from rest positions
436
437             How good of a metric this is remains to be seen. It is ↩
                    just the
438             total displacement of all crowns from their axial rest ↩
                    positions.
439             """
440             dists = np.diff(np.hstack([0, self.axial]))
441             return np.sum(np.abs(dists - self.rests))
442
443         def transition(self):
444             """Give each cross-bridge in the filament a chance to ↩
                    transition"""
445             transitions = [crown.transition() for crown in self.crowns]
446             return transitions
447
448         def get_axial_location(self, index):
449             """Return the axial location at the given crown index"""
450             return self.axial[index]
451
452         def _set_timestep(self, timestep):
453             """Set the length of time step used to calculate ↩
                    transitions"""
454             [crown._set_timestep(timestep) for crown in self.crowns]
455
456         def get_states(self):
457             """Return the numeric states (0,1,2) of each face's ↩
                    cross-bridges"""
```

```
458            return [face.get_states() for face in self.thick_faces]
459
460        def get_lattice_spacing(self):
461            """Return the lattice's spacing"""
462            return self.parent_lattice.get_lattice_spacing()
463
464        def _axial_thick_filament_forces(self, axial_locations=None):
465            """The axial force generated by the thick filament at each ↩
                  crown
466
467            This returns the axial force at each thick filament ↩
                  location,
468            not counting any force generated by bound cross-bridge ↩
                  heads.
469            Parameters:
470                axial_locations: location of each crown (optional)
471            Return:
472                forces: axial force of the thick filament at each crown
473            """
474            # Use the thick filament's stored axial locations if none ↩
                  are passed
475            if axial_locations == None:
476                axial_locations = np.hstack([0, self.axial])
477            else:
478                axial_locations = np.hstack([0, axial_locations])
479            # Find the distance from crown to crown, then the ↩
                  resulting forces
480            dists = np.diff(axial_locations)
481            spring_force = (dists - self.rests) * self.k
482            spring_force = np.hstack([spring_force, 0]) # Last node ↩
                  not connected
483            net_force_at_crown = np.diff(spring_force)
484            return net_force_at_crown
485
486        @staticmethod
487        def _radial_force_to_components(crown_force, orientation):
488            """Convert radial components of a crown's force into a y,z ↩
                  vector
489
490            Myosin crowns come in two varieties, types a and b. They are
491            oriented thusly:
492                Type A          Type B
493              a0      a1          a0
494                  mf
495                                   mf
496                  a2          a2      a1
497            The purpose of this function is to sort the force a single ↩
                  crown
498            generates out into a single (y,z) vector.
499            Parameters:
```

```
500                 crown_force: force a single crown generates, (f_a0, ↩
                        f_a1, f_a2)
501                 orientation: that crown's orientation; 0 for type A, 1 ↩
                        for B
502             Returns:
503                 force: the force the crown generates, (y, z)
504             """
505             if orientation == 0:
506                 f_y = -0.866 * crown_force[0] + 0.866 * crown_force[1]
507                 f_z = 0.5 * crown_force[0] + 0.5 * crown_force[1] - ↩
                        crown_force[2]
508             else:
509                 f_y = 0.866 * crown_force[1] - 0.866 * crown_force[2]
510                 f_z = crown_force[0] - 0.5 * crown_force[1] - 0.5 * ↩
                        crown_force[2]
511             return (f_y, f_z)
512
513
514 if __name__ == '__main__':
515     print("mf.py is really meant to be called as a supporting ↩
            module")
```

B.3 File mh.py

```
1   #!/usr/bin/env python
2   # encoding: utf-8
3   """
4   mh.py - A single myosin head
5
6   Created by Dave Williams on 2010-01-04.
7   """
8
9   import numpy.random as random
10  random.seed() # Ensure proper seeding
11  from numpy import pi, sqrt, log, radians
12  import math as m
13  import warnings
14
15  class Spring(object):
16      """A generic spring, from which we make the myosin heads"""
17      def __init__(self, config):
18          ## Passed variables
19          self.r_w = config['rest_weak']
20          self.r_s = config['rest_strong']
21          self.k_w = config['konstant_weak']
22          self.k_s = config['konstant_strong']
23          ## Diffusion governors
24          # k_T = Boltzmann constant * temperature = (1.381E-23 J/K ↩
                * 288 K)
```

```
25          k_t = 1.381*10**-23 * 288 * 10**21 #10**21 converts J to ↩
                pN*nM
26          # Normalize: a factor used to normalize the PDF of the ↩
                segment values
27          self.normalize = sqrt(2*pi*k_t/self.k_w)
28          self.stand_dev = sqrt(k_t/self.k_w) # of segment values
29
30      def rest(self, state):
31          """Return the rest value of the spring in state state
32
33          Takes:
34              state: the state of the spring, ['free'|'loose'|'tight']
35          Returns:
36              length/angle: rest length/angle of the spring in the ↩
                    given state
37          """
38          if state in ("free", "loose"):
39              return self.r_w
40          elif state == "tight":
41              return self.r_s
42          else:
43              warnings.warn("Improper value for spring state")
44
45      def constant(self, state):
46          """Return the spring constant of the spring in state state
47
48          Takes:
49              state: the state of the spring, ['free'|'loose'|'tight']
50          Returns:
51              spring constant: for the spring in the given state
52          """
53          if state in ("free", "loose"):
54              return self.k_w
55          elif state == "tight":
56              return self.k_s
57          else:
58              warnings.warn("Improper value for spring state")
59
60      def energy(self, spring_val, state):
61          """Given a current length/angle, return stored energy
62
63          Takes:
64              spring_val: a spring length or angle
65              state: a spring state, ['free'|'loose'|'tight']
66          Returns:
67              energy: the energy required to achieve the given value
68          """
69          if state in ("free", "loose"):
70              return (0.5 * self.k_w * m.pow((spring_val-self.r_w), ↩
                    2))
```

```
71          elif state == "tight":
72              return (0.5 * self.k_s * m.pow((spring_val-self.r_s), ↩
                    2))
73          else:
74              warnings.warn("Improper value for spring state")
75
76      def bop(self):
77          """Bop for a new value, given an exponential energy dist
78
79          A longer explanation is in singlexb/Crossbridge.py
80          Takes:
81              nothing: assumes the spring to be in the unbound state
82          Returns:
83              spring_value: the length or angle of the spring after ↩
                    diffusion"""
84          return (random.normal(self.r_w, self.stand_dev))
85
86
87  class SingleSpringHead(object):
88      """A single-spring myosin head, as in days of yore"""
89      def __init__(self):
90          """Create the spring that makes up the head and set energy ↩
                values"""
91          self.state = "free"
92          self.g = Spring({
93              'rest_weak': 5,
94              'rest_strong': 0,
95              'konstant_weak': 5 / 3.976,
96              'konstant_strong': 5 / 3.976})
97          # Free energy calculation helpers
98          g_atp = 13 # In units of RT
99          atp = 5  * 10**-3
100         adp = 30 * 10**-6
101         phos  = 3  * 10**-3
102         self.deltaG = abs(-g_atp - log(atp / (adp * phos)))
103         self.alpha = 0.28
104         self.eta = 0.68
105         # The time-step, master of all time
106         self.timestep = 1 # ms
107
108     def transition(self, bs):
109         """Transition to a new state (or not)
110
111         Takes:
112             bs: relative Crown to Actin distance (x,y)
113         Returns:
114             boolean: transition that occurred (as string) or None
115         """
116         ## Transitions rates are checked against a random number
117         check = random.rand()
```

```
118         ## Check for transitions depending on the current state
119         if self.state == "free":
120             if self._r12(bs) > check:
121                 self.state = "loose"
122                 return '12'
123         elif self.state == "loose":
124             if self._r23(bs) > check:
125                 self.state = "tight"
126                 return '23'
127             elif self._r21(bs) > check:
128                 self.state = "free"
129                 return '21'
130         elif self.state == "tight":
131             if self._r31(bs) > check:
132                 self.state = "free"
133                 return '31'
134             elif self._r32(bs) > check:
135                 self.state = "loose"
136                 return '32'
137         # Got this far? Than no transition occurred!
138         return None
139 
140     def axialforce(self, tip_location):
141         """Find the axial force a Head generates at a given location
142 
143         Takes:
144             tip_location: relative Crown to Actin distance (x,y)
145         Returns:
146             f_x: the axial force generated by the Head
147         """
148         ## Get the Head length
149         g_len = tip_location[0]
150         ## Write all needed values to local variables
151         g_s = self.g.rest(self.state)
152         g_k = self.g.constant(self.state)
153         ## Find and return force
154         f_x = g_k * (g_len - g_s)
155         return f_x
156 
157     def radialforce(self, tip_location):
158         """Find the radial force a Head generates at a given ↩
               location
159 
160         Takes:
161             tip_location: relative Crown to Actin distance (x,y)
162         Returns:
163             f_y: the radial force generated by the Head
164         """
165         return 0.0
166 
```

```
167     def energy(self, tip_location, state=None):
168         """Return the energy in the xb with the given parameters
169 
170         Takes:
171             tip_location: relative Crown to Actin distance (x,y)
172             state: kinetic state of the cross-bridge, ↩
                   ['free'|'loose'|'tight']
173         Returns:
174             xb_energy: the energy stored in the cross-bridge"""
175         if state is None:
176             state = self.state
177         return self.g.energy(tip_location[0], state)
178 
179     def get_numeric_state(self):
180         """Return the numeric state (0, 1, or 2) of the head"""
181         lookup_state = {"free":0, "loose":1, "tight":2}
182         return lookup_state[self.state]
183 
184     def _set_timestep(self, timestep):
185         """Set the length of time step used to calculate ↩
              transitions"""
186         self.timestep = timestep
187 
188     def _r12(self, bs):
189         """Binding rate, based on the distance from the Head tip ↩
              to a Actin
190 
191         Takes:
192             bs: relative Crown to Actin distance (x,y)
193         Returns:
194             probability: chance of binding occurring
195         """
196         ## Get needed values
197         k_xb = self.g.constant("free")
198         xb_0 = self.g.rest("free")
199         A = 2000  # From Tanner, 2008 Pg 1209
200         ## Calculate the binding probability
201         rate = (A * sqrt(k_xb / (2 * pi)) *
202                 m.exp(-.5 * k_xb * (bs[0] - xb_0)**2)) * ↩
                    self.timestep
203         return float(rate)
204 
205     def _r21(self, bs):
206         """The reverse transition, from loosely bound to unbound
207 
208         Takes:
209             bs: relative Crown to Actin distance (x,y)
210         Returns:
211             rate: probability of transition occurring this timestep
212         """
```

```
213         ## The rate depends on the states' free energies
214         g_1 = self._free_energy(bs, "free")
215         g_2 = self._free_energy(bs, "loose")
216         ## Rate, as in pg 1209 of Tanner et al, 2007
217         rate = self._r12(bs) / m.exp(g_1 - g_2)
218         return float(rate)
219
220     def _r23(self, bs):
221         """Probability of becoming tightly bound if loosely bound
222
223         Takes:
224             bs: relative Crown to Actin distance (x,y)
225         Returns:
226             rate: probability of becoming tightly bound
227         """
228         ## Get other needed values
229         k_xb = self.g.constant("loose")
230         xb_0 = self.g.rest("loose")
231         B = 100   # From Tanner, 2008 Pg 1209
232         C = 1
233         D = 1
234         ## Rate taken from single cross-bridge work
235         rate = (B / sqrt(k_xb) * (1 - m.tanh(C * sqrt(k_xb) *
236                 (bs[0] - xb_0))) + D) * self.timestep
237         return float(rate)
238
239     def _r32(self, bs):
240         """The reverse transition, from tightly to loosely bound
241
242         Takes:
243             bs: relative Crown to Actin distance (x,y)
244         Returns:
245             rate: probability of becoming loosely bound
246         """
247         ## Governed as in self_r21
248         g_2 = self._free_energy(bs, "loose")
249         g_3 = self._free_energy(bs, "tight")
250         rate = self._r23(bs) / m.exp(g_2 - g_3)
251         return float(rate)
252
253     def _r31(self, bs):
254         """Probability of unbinding if tightly bound
255
256         Takes:
257             bs: relative Crown to Actin distance (x,y)
258         Returns:
259             rate: probability of detaching from the binding site
260         """
261         ## Get needed values
262         k_xb = self.g.constant("tight")
```

```
263         M = 3600 # From Tanner, 2008 Pg 1209
264         N = 40
265         P = 20
266         ## Based on the energy in the tight state
267         rate = (sqrt(k_xb) * (sqrt(M * (bs[0]-4.76)**2) -
268                 N * (bs[0]-4.76)) + P) * self.timestep
269         return float(rate)
270
271     def _free_energy(self, tip_location, state):
272         """Free energy of the Head
273
274         Takes:
275             tip_location: relative Crown to Actin distance (x,y)
276             state: kinetic state of the cross-bridge, ↩
                    ['free'|'loose'|'tight']
277         Returns:
278             energy: free energy of the head in the given state
279         """
280         if state == "free":
281             return 0
282         elif state == "loose":
283             k_xb = self.g.constant(state)
284             xb_0 = self.g.rest(state)
285             x = tip_location[0]
286             return self.alpha * -self.deltaG + k_xb * (x - xb_0)**2
287         elif state == "tight":
288             k_xb = self.g.constant(state)
289             x = tip_location[0]
290             return self.eta * -self.deltaG + k_xb * x**2
291
292
293 class Head(object):
294     """Head implements a single myosin head"""
295     def __init__(self):
296         """Create the springs that make up the head and set energy ↩
                values"""
297         # Remember thine kinetic state
298         self.state = "free"
299         # Create the springs which make up the head
300         self.c = Spring({
301             'rest_weak': radians(47.16),
302             'rest_strong': radians(73.20),
303             'konstant_weak': 40,
304             'konstant_strong': 40})
305         self.g = Spring({
306             'rest_weak': 19.93,
307             'rest_strong': 16.47,
308             'konstant_weak': 2,
309             'konstant_strong': 2})
310         # Free energy calculation helpers
```

```
311         g_atp = 13 # In units of RT
312         atp = 5  * 10**-3
313         adp = 30 * 10**-6
314         phos  = 3  * 10**-3
315         deltaG = abs(-g_atp - log(atp / (adp * phos)))
316         self.alphaDG = 0.28 * -deltaG
317         self.etaDG = 0.68 * -deltaG
318         # The time-step, master of all time
319         self.timestep = 1 # ms
320 
321     def transition(self, bs):
322         """Transition to a new state (or not)
323 
324         Takes:
325             bs: relative Crown to Actin distance (x,y)
326         Returns:
327             boolean: transition that occurred (as string) or None
328         """
329         ## Transitions rates are checked against a random number
330         check = random.rand()
331         ## Check for transitions depending on the current state
332         if self.state == "free":
333             if self._bind(bs) > check:
334                 self.state = "loose"
335                 return '12'
336         elif self.state == "loose":
337             if self._r23(bs) > check:
338                 self.state = "tight"
339                 return '23'
340             elif self._r21(bs) > check:
341                 self.state = "free"
342                 return '21'
343         elif self.state == "tight":
344             if self._r31(bs) > check:
345                 self.state = "free"
346                 return '31'
347             elif self._r32(bs) > check:
348                 self.state = "loose"
349                 return '32'
350         # Got this far? Than no transition occurred!
351         return None
352 
353     def axialforce(self, tip_location):
354         """Find the axial force a Head generates at a given location
355 
356         Takes:
357             tip_location: relative Crown to Actin distance (x,y)
358         Returns:
359             f_x: the axial force generated by the Head
360         """
```

```
361         ## Get the Head length and angle
362         (c_ang, g_len) = self._seg_values(tip_location)
363         ## Write all needed values to local variables
364         c_s = self.c.rest(self.state)
365         g_s = self.g.rest(self.state)
366         c_k = self.c.constant(self.state)
367         g_k = self.g.constant(self.state)
368         ## Find and return force
369         f_x = (g_k * (g_len - g_s) * m.cos(c_ang) +
370                1/g_len * c_k * (c_ang - c_s) * m.sin(c_ang))
371         return f_x
372 
373     def radialforce(self, tip_location):
374         """Find the radial force a Head generates at a given ↩
                location
375 
376         Takes:
377             tip_location: relative Crown to Actin distance (x,y)
378         Returns:
379             f_y: the radial force generated by the Head
380         """
381         ## Get the Head length and angle
382         (c_ang, g_len) = self._seg_values(tip_location)
383         ## Write all needed values to local variables
384         c_s = self.c.rest(self.state)
385         g_s = self.g.rest(self.state)
386         c_k = self.c.constant(self.state)
387         g_k = self.g.constant(self.state)
388         ## Find and return force
389         f_y = (g_k * (g_len - g_s) * m.sin(c_ang) +
390                1/g_len * c_k * (c_ang - c_s) * m.cos(c_ang))
391         return f_y
392 
393     def energy(self, tip_location, state=None):
394         """Return the energy in the xb with the given parameters
395 
396         Takes:
397             tip_location: relative Crown to Actin distance (x,y)
398             state: kinetic state of the cross-bridge, ↩
                    ['free'|'loose'|'tight']
399         Returns:
400             xb_energy: the energy stored in the cross-bridge"""
401         if state == None:
402             state = self.state
403         (ang, dist) = self._seg_values(tip_location)
404         xb_energy = self.c.energy(ang, state) + ↩
                self.g.energy(dist, state)
405         return xb_energy
406 
407     def get_numeric_state(self):
```

```
408         """Return the numeric state (0, 1, or 2) of the head"""
409         lookup_state = {"free":0, "loose":1, "tight":2}
410         return lookup_state[self.state]
411 
412     def _set_timestep(self, timestep):
413         """Set the length of time step used to calculate ↩
            transitions"""
414         self.timestep = timestep
415 
416     def _bind(self, bs):
417         """Bind (or don't) based on the distance from the Head tip ↩
            to a Actin
418 
419         Takes:
420             bs: relative Crown to Actin distance (x,y)
421         Returns:
422             probability: chance of binding occurring
423         """
424         ## Flag indicates successful diffusion
425         bop_right = False
426         while bop_right is False:
427             ## Bop the springs to get new values
428             c_ang = self.c.bop()
429             g_len = self.g.bop()
430             ## Translate those values to an (x,y) position
431             tip = (g_len * m.cos(c_ang), g_len * m.sin(c_ang))
432             ## Only a bop that lands short of the thin fil is valid
433             bop_right = bs[1] >= tip[1]
434         ## Find the distance to the binding site
435         distance = m.hypot(bs[0]-tip[0], bs[1]-tip[1])
436         ## The binding prob is dependent on the exp of the dist
437         # Prob = \tau * \exp^{-dist^2} * timestep
438         probability = 72 * m.exp(-distance**2) * self.timestep
439         ## Return the probability
440         return probability
441 
442     def _r21(self, bs):
443         """The reverse transition, from loosely bound to unbound
444 
445         This depends on the rate r12, the binding rate, which is ↩
            given
446         in a stochastic manner. Thus _r21 is returning not the ↩
            rate of
447         going from loosely bound to tightly bound, but the change ↩
            that
448         occurs in one particular timestep, the stochastic rate.
449         Takes:
450             bs: relative Crown to Actin distance (x,y)
451         Returns:
452             rate: probability of transition occurring this timestep
```

```
453         """
454         ## The rate depends on the states' free energies
455         unbound_free_energy = self._free_energy(bs, "free")
456         loose_free_energy = self._free_energy(bs, "loose")
457         ## Rate, as in pg 1209 of Tanner et al, 2007
458         ## With added reduced-detachment factor, increases dwell ↩
                time
459         rate = self._bind(bs) / m.exp(unbound_free_energy - ↩
                loose_free_energy)
460         return float(rate)
461
462     def _r23(self, bs):
463         """Probability of becoming tightly bound if loosely bound
464
465         Takes:
466             bs: relative Crown to Actin distance (x,y)
467         Returns:
468             rate: probability of becoming tightly bound
469         """
470         ## The transition rate depends on state energies
471         loose_energy = self.energy(bs, "loose")
472         tight_energy = self.energy(bs, "tight")
473         ## Rate taken from single cross-bridge work
474         #rate = (.1 * (1 + m.tanh(.4 * (loose_energy - ↩
                tight_energy) + 4)) \
475         #         +.001) * self.timestep
476         rate = 10*(.1 * (1 + m.tanh(.4 * (loose_energy - ↩
                tight_energy) + 4)) \
477                 +.001) * self.timestep
478         return float(rate)
479
480     def _r32(self, bs):
481         """The reverse transition, from tightly to loosely bound
482
483         Takes:
484             bs: relative Crown to Actin distance (x,y)
485         Returns:
486             rate: probability of becoming loosely bound
487         """
488         ## Governed as in self_r21
489         loose_free_energy = self._free_energy(bs, "loose")
490         tight_free_energy = self._free_energy(bs, "tight")
491         rate = self._r23(bs)/ m.exp(loose_free_energy - ↩
                tight_free_energy)
492         return float(rate)
493
494     def _r31(self, bs):
495         """Probability of unbinding if tightly bound
496
497         Takes:
```

```
498                 bs: relative Crown to Actin distance (x,y)
499             Returns:
500                 rate: probability of detaching from the binding site
501             """
502             ## Based on the energy in the tight state
503             tight_energy = self.energy(bs, "tight")
504             rate =  (m.sqrt(.01 * tight_energy) + 0.02) * self.timestep
505             return float(rate)
506 
507         def _free_energy(self, tip_location, state):
508             """Free energy of the Head
509 
510             Takes:
511                 tip_location: relative Crown to Actin distance (x,y)
512                 state: kinetic state of the cross-bridge, ↩
                        ['free'|'loose'|'tight']
513             Returns:
514                 energy: free energy of the head in the given state
515             """
516             if state == "free":
517                 return 0
518             elif state == "loose":
519                 return self.alphaDG + self.energy(tip_location, state)
520             elif state == "tight":
521                 return self.etaDG + self.energy(tip_location, state)
522 
523         @staticmethod
524         def _seg_values(tip_location):
525             """Return the length and angle to the Head tip
526 
527             Takes:
528                 tip_location: relative Crown to Actin distance (x,y)
529             Returns:
530                 (c_ang, g_len): the angle and length of the Head's ↩
                        springs
531             """
532             c_ang = m.atan2(tip_location[1], tip_location[0])
533             g_len = m.hypot(tip_location[1], tip_location[0])
534             return (c_ang, g_len)
535 
536 
537     class Crossbridge(Head):
538         """A cross-bridge, including status of links to actin sites"""
539         def __init__(self, face_index, face_parent, thin_face):
540             """Set up the cross-bridge
541 
542             Parameters:
543                 face_index: the cross-bridge's index on the parent face
544                 face_parent: the associated thick filament face
545                 thin_face: the face instance opposite this cross-bridge
```

```
546         """
547         # Do that super() voodoo that instantiates the parent Head
548         super(Crossbridge, self).__init__()
549         # What is your name, where do you sit on the parent face?
550         self.face_index = face_index
551         # What log are you a bump upon?
552         self.face_parent = face_parent
553         # Remember who thou art squaring off against
554         self.thin_face = thin_face
555         # Remember if thou art bound unto an actin
556         self.bound_to = None # None if unbound, BindingSite object ↩
                otherwise
557 
558     def __repr__(self):
559         """String representation of the cross-bridge"""
560         out = '__XB_%02d__State_%s__Forces_%d_%d__'%(
561             self.face_index, self.state,
562             self.axialforce(), self.radialforce())
563         #out = '__XB_%02d_on_Face_%1d__ \n'%(
564         #         self.face_index, self.face_parent.orientation)
565         #if self.bound_to is None:
566         #    out += 'State: Unbound \n'
567         #else:
568         #    out += 'State: Bound, %s \n'%super(Crossbridge, ↩
                self).state
569         #    out += 'Forces: %d/%d pN (ax/rad) ↩
                \n'%(self.axialforce, self.radialforce)
570         return out
571 
572     def transition(self):
573         """Gather the needed information and try a transition
574 
575         Parameters:
576             None
577         Returns:
578             transition: string of transition ('12', '32', etc.) or ↩
                    None
579         """
580         # When unbound, try to bind, otherwise just try a transition
581         if self.bound_to is None:
582             # Find the lattice spacing
583             lattice_spacing = self._get_lattice_spacing()
584             # Find this cross-bridge's axial location
585             xb_axial_loc = self._get_axial_location()
586             # Find the potential binding site
587             actin_site = self.thin_face.nearest(xb_axial_loc)
588             actin_axial_loc = actin_site.get_axial_location()
589             # Find the axial separation
590             axial_sep = actin_axial_loc - xb_axial_loc
591             # Combine the two distances
```

```
592                 distance_to_site = (axial_sep, lattice_spacing)
593                 # Allow the myosin head to take it from here
594                 trans = super(Crossbridge, ↩
                        self).transition(distance_to_site)
595                 # Process changes to bound state
596                 if trans == '12':
597                     self.bound_to = actin_site
598                     actin_site.bind_to(self)
599                 else:
600                     assert (trans is None), 'Bound state mismatch'
601             else:
602                 # Get the distance to the actin site
603                 distance_to_site = self._dist_to_bound_actin()
604                 # Allow the myosin head to take it from here
605                 trans = super(Crossbridge, ↩
                        self).transition(distance_to_site)
606                 # Process changes to the bound state
607                 if trans in set(('21', '31')):
608                     self.bound_to.bind_to(None)
609                     self.bound_to = None
610                 else:
611                     assert (trans in set(('23', '32', None))) , 'State ↩
                            mismatch'
612             return trans
613 
614         def axialforce(self, base_axial_loc=None, tip_axial_loc = None):
615             """Gather needed information and return the axial force
616 
617             Parameters:
618                 base_axial_location: location of the crown (optional)
619                 tip_axial_loc: location of an attached actin node ↩
                        (optional)
620             Returns:
621                 f_x: the axial force generated by the cross-bridge
622             """
623             # Unbound? No force!
624             if self.bound_to is None:
625                 return 0.0
626             # Else, get the distance to the bound site and run with it
627             distance = self._dist_to_bound_actin(base_axial_loc, ↩
                    tip_axial_loc)
628             # Allow the myosin head to take it from here
629             return super(Crossbridge, self).axialforce(distance)
630 
631         def radialforce(self):
632             """Gather needed information and return the radial force
633 
634             Parameters:
635                 None
636             Returns:
```

```
637                 f_y: the radial force generated by the cross-bridge
638             """
639             # Unbound? No force!
640             if self.bound_to is None:
641                 return 0.0
642             # Else, get the distance to the bound site and run with it
643             distance_to_site = self._dist_to_bound_actin()
644             # Allow the myosin head to take it from here
645             return super(Crossbridge, ↩
                    self).radialforce(distance_to_site)
646 
647         def _get_axial_location(self):
648             """Find the axial location of the thick filament ↩
                    attachment point
649 
650             Parameters:
651                 None
652             Returns:
653                 axial: the axial location of the cross-bridge base
654             """
655             axial = self.face_parent.get_axial_location(self.face_index)
656             return axial
657 
658         def _dist_to_bound_actin(self, xb_axial_loc=None, ↩
                tip_axial_loc=None):
659             """Find the (x,y) distance to the bound actin
660 
661             This is the distance format used by the myosin head.
662             Parameters:
663                 xb_axial_loc: current axial location of the crown ↩
                        (optional)
664                 tip_axial_loc: location of an attached actin node ↩
                        (optional)
665             Returns:
666                 (x,y): the axial distance between the cross-bridge ↩
                        base and
667                        the actin site (x), and the lattice spacing (y)
668             """
669             # Are you really bound?
670             assert (self.bound_to is not None) , "Lies, you're unbound!"
671             # Find the lattice spacing
672             lattice_spacing = self._get_lattice_spacing()
673             # Find this cross-bridge's axial location if need be
674             if xb_axial_loc is None:
675                 xb_axial_loc = self._get_axial_location()
676             # Find the distance to the bound actin site if need be
677             if tip_axial_loc is None:
678                 tip_axial_loc = self.bound_to.get_axial_location()
679             # Combine the two distances
680             return (tip_axial_loc - xb_axial_loc, lattice_spacing)
```

```
681
682     def _get_lattice_spacing(self):
683         """Ask our superiors for lattice spacing data"""
684         return self.face_parent.get_lattice_spacing()
685
686
687 if __name__ == '__main__':
688     print("mh.py is really meant to be called as a supporting ↩
           module")
```

B.4 File hs.py

```
 1 #!/usr/bin/env python
 2 # encoding: utf-8
 3 """
 4 hs.py - A half-sarcomere model with multiple thick and thin ↩
       filaments
 5
 6 Created by Dave Williams on 2009-12-31.
 7 """
 8
 9 import sys
10 import os
11 import multiprocessing as mp
12 import unittest
13 import time
14 import numpy as np
15 import scipy.optimize as opt
16 import af
17 import mf
18
19 class hs(object):
20     """The half-sarcomere and ways to manage it"""
21     def __init__(self, lattice_spacing=None, z_line=None):
22         """ Create the data structure that is the half-sarcomere ↩
               model
23
24         Parameters:
25             lattice_spacing: the surface-to-surface distance (14.0)
26             z_line: the length of the half-sarcomere (1250)
27         Returns:
28             None
29
30         This is the organizational basis that the rest of the ↩
               model, and
31         classes representing the other structures therein will use ↩
               when
32         running. It contains the following properties:
33
```

```
34         ## Half-sarcomere properties: these are properties that ↩
              can be
35         interpreted as belonging to the overall model, not to any ↩
              thick or
36         thin filament.
37
38         lattice_spacing:
39             the face to face lattice spacing for the whole model
40         m_line:
41             x axis location of the m line
42         h_line:
43             x axis location of the h line
44         hiding_line:
45             x axis location below which actin sites are hidden
46
47         ## Thick Filament Properties: each is a tuple of thick ↩
              filaments
48         (filament_0, filament_1, filament_2, filament_3) where each
49         filament_x is giving the actual properties of that ↩
              particular
50         filament.
51
52         thick_location:
53             each tuple location is a list of x axis locations
54         thick_crowns:
55             each tuple location is a tuple of links to crown ↩
                  instances
56         thick_link:
57             each tuple location is a list consisting of three (one ↩
                  for each
58             myosin head in the crown) of either None or a link to ↩
                  a thin_site
59         thick_adjacent:
60             each tuple location is a tuple of links to adjacent ↩
                  thin filaments
61         thick_face:
62             each tuple location is a tuple of length six, each ↩
                  location of
63             which contains a tuple of links to myosin heads that ↩
                  are facing
64             each surrounding thin filament
65         thick_bare_zone:
66             a single value, the length of each filament before the ↩
                  first crown
67         thick_crown_spacing:
68             a single value, the distance between two crowns on a ↩
                  single filament
69         thick_k:
70             a single value, the spring constant of the thick ↩
                  filament between
```

```
71                 any given pair of crowns
72
73             ## Thin Filament Properties: arranged in the same manner ↩
                   as the
74             thick filament properties, but for the eight thin filaments
75
76             thin_location:
77                 each tuple location is a list of x axis locations
78             thin_link:
79                 each tuple location is a list consisting of entries ↩
                       (one for each
80                 thin_site on the thin_filament) of either a None or a ↩
                       link to a
81                 thick_crown
82             thin_adjacent:
83                 each tuple location is a tuple of links to adjacent ↩
                       thick filaments
84             thin_face:
85                 each tuple location is a tuple of length three, each ↩
                       location of
86                 which contains a tuple of links to thin filament sites ↩
                       that are
87                 facing each surrounding thick filament
88             thin_site_spacing:
89                 the axial distance from one thin filament binding site ↩
                       to another
90             thin_k:
91                 a single value, the spring constant of the thin ↩
                       filament between
92                 any given pair of thin binding sites
93
94             """
95             # Default LS and Z-line, using None to simplify calling
96             if lattice_spacing is None:
97                 lattice_spacing = 14.0
98             if z_line is None:
99                 z_line = 1250
100            # Store these values for posterity
101            self.lattice_spacing = lattice_spacing
102            self.z_line = z_line
103            # Track how long we've been running
104            self.current_timestep = 0
105            # Create the thin filaments, unlinked but oriented on ↩
                   creation.
106            thin_orientations = ([4,0,2], [3,5,1], [4,0,2], [3,5,1],
107                    [3,5,1], [4,0,2], [3,5,1], [4,0,2])
108            self.thin = tuple([af.ThinFilament(self, orientation, ↩
                   z_line) for
109                orientation in thin_orientations])
110            # Determine the hiding line
```

```
111         self.set_hiding_line()
112         # Create the thick filaments, remembering they are ↩
               arranged thus:
113         # ----------------------------
114         # |   Actin around myosin    |
115         # |--------------------------|
116         # |      a1      a3          |
117         # |  a0      a2      a0      |
118         # |      M0      M1          |
119         # |  a4      a6      a4      |
120         # |      a5      a7      a5  |
121         # |          M2      M3      |
122         # |      a1      a3      a1  |
123         # |          a2      a0      |
124         # ----------------------------
125         # and that when choosing which actin face to link to which ↩
               thick
126         # filament face, use these face orders:
127         # ----------------------------------------------------
128         # | Myosin face order  |       Actin face order      |
129         # |--------------------|-----------------------------|
130         # |        a1          |                             |
131         # |    a0      a2      |  m0      m1          m0     |
132         # |        mf          |      af      OR             |
133         # |    a5      a3      |                      af     |
134         # |        a4          |      m2          m2      m1 |
135         # ----------------------------------------------------
136         self.thick = (
137                 mf.ThickFilament(self, (
138                     self.thin[0].thin_faces[1], ↩
                           self.thin[1].thin_faces[2],
139                     self.thin[2].thin_faces[2], ↩
                           self.thin[6].thin_faces[0],
140                     self.thin[5].thin_faces[0], ↩
                           self.thin[4].thin_faces[1]),
141                     1),
142                 mf.ThickFilament(self, (
143                     self.thin[2].thin_faces[1], ↩
                           self.thin[3].thin_faces[2],
144                     self.thin[0].thin_faces[2], ↩
                           self.thin[4].thin_faces[0],
145                     self.thin[7].thin_faces[0], ↩
                           self.thin[6].thin_faces[1]),
146                     1),
147                 mf.ThickFilament(self, (
148                     self.thin[5].thin_faces[1], ↩
                           self.thin[6].thin_faces[2],
149                     self.thin[7].thin_faces[2], ↩
                           self.thin[3].thin_faces[0],
```

```
150                         self.thin[2].thin_faces[0], ↩
                               self.thin[1].thin_faces[1]),
151                         1),
152                     mf.ThickFilament(self, (
153                         self.thin[7].thin_faces[1], ↩
                               self.thin[4].thin_faces[2],
154                         self.thin[5].thin_faces[2], ↩
                               self.thin[1].thin_faces[0],
155                         self.thin[0].thin_faces[0], ↩
                               self.thin[3].thin_faces[1]),
156                         1)
157                     )
158             # Now the thin filaments need to be linked to thick ↩
                   filaments, use
159             # the face orders from above and the following arrangement:
160             # ----------------------------
161             # |    Myosin around actin     |
162             # |--------------------------|
163             # |       m3       m2       m3   |
164             # |           A1       A3        |
165             # |       A0       A2            |
166             # |   m1       m0       m1       |
167             # |       A4       A6            |
168             # |           A5       A7        |
169             # |       m3       m2       m3   |
170             # ----------------------------
171             # The following may be hard to read, but it has been ↩
                   checked and
172             # may be moderately trusted. CDW-20100406
173             self.thin[0].set_thick_faces((self.thick[3].thick_faces[4],
174                 self.thick[0].thick_faces[0], ↩
                       self.thick[1].thick_faces[2]))
175             self.thin[1].set_thick_faces((self.thick[3].thick_faces[3],
176                 self.thick[2].thick_faces[5], ↩
                       self.thick[0].thick_faces[1]))
177             self.thin[2].set_thick_faces((self.thick[2].thick_faces[4],
178                 self.thick[1].thick_faces[0], ↩
                       self.thick[0].thick_faces[2]))
179             self.thin[3].set_thick_faces((self.thick[2].thick_faces[3],
180                 self.thick[3].thick_faces[5], ↩
                       self.thick[1].thick_faces[1]))
181             self.thin[4].set_thick_faces((self.thick[1].thick_faces[3],
182                 self.thick[0].thick_faces[5], ↩
                       self.thick[3].thick_faces[1]))
183             self.thin[5].set_thick_faces((self.thick[0].thick_faces[4],
184                 self.thick[2].thick_faces[0], ↩
                       self.thick[3].thick_faces[2]))
185             self.thin[6].set_thick_faces((self.thick[0].thick_faces[3],
186                 self.thick[1].thick_faces[5], ↩
                       self.thick[2].thick_faces[1]))
```

```
187         self.thin[7].set_thick_faces((self.thick[1].thick_faces[4],
188             self.thick[3].thick_faces[0], ↩
                self.thick[2].thick_faces[2]))
189 
190     def run(self, time_steps=100, callback=None, bar=True):
191         """Run the model for the specified number of timesteps
192 
193         Parameters:
194             time_steps: number of time steps to run the model for ↩
                (100)
195             callback: function to be executed after each time step ↩
                to
196                 collect data. The callback function takes the ↩
                    sarcomere
197                 in its current state as its only argument. ↩
                    (Defaults to
198                 the axial force at the M-line if not specified.)
199             bar: progress bar control,False means don't display, ↩
                True
200                 means give us the basic progress reports, if a ↩
                    function
201                 is passed, it will be called as f(completed_steps,
202                 total_steps, sec_left, sec_passed, process_name).
203                 (Defaults to True)
204         Returns:
205             output: the results of the callback after each timestep
206         """
207         # Callback defaults to the axial force at the M-line
208         if callback is None:
209             callback = lambda sarc: sarc.axialforce()
210         # Create a place to store callback information and note ↩
            the time
211         output = []
212         tic = time.time()
213         # Run through each timestep
214         for i in range(time_steps):
215             self.timestep()
216             output.append(callback(self))
217             # Update us on how it went
218             toc = int((time.time()-tic) / (i+1) * (time_steps-i-1))
219             proc_name = mp.current_process().name
220             if bar == True:
221                 sys.stdout.write("\n" + proc_name +
222                     " finished timestep %i of %i, %ih%im%is left"\
223                     %(i+1, time_steps, toc/60/60, toc/60%60, ↩
                        toc%60))
224                 sys.stdout.flush()
225             elif type(bar) == type(lambda x:x):
226                 bar(i, time_steps, toc, time.time()-tic, proc_name)
227         return output
```

```
228
229     def timestep(self):
230         """Move the model one step forward in time, allowing the
231         myosin heads a chance to bind and then balancing forces
232         """
233         # Record our passage through time
234         self.current_timestep += 1
235         # Update bound states
236         self.last_transitions = [thick.transition() for thick in ↩
                self.thick]
237         # Balance forces
238         # TODO Implement solution caching to speed calculations
239         dist = self.balance()
240         return dist
241
242     def axialforce(self):
243         """Sum of each thick filament's axial force on the M-line ↩
                """
244         return sum([thick.effective_axial_force() for thick in ↩
                self.thick])
245
246     def radialtension(self):
247         """The sum of the thick filaments' radial tensions"""
248         return sum([t.radialtension() for t in self.thick])
249
250     def radialforce(self):
251         """The sum of the thick filaments' radial forces, as a ↩
                (y,z) vector"""
252         return np.sum([t.radial_force_of_filament() for t in ↩
                self.thick], 0)
253
254     def balance(self):
255         """Manage the movement of the axial locations to minimize ↩
                energy
256
257         Parameters:
258             None
259         Returns:
260             dist: the distances moved by each node
261         """
262         # Create the terrible giant location list
263         all_axial_locs = self.flatten_locations()
264         # Balance forces
265         soln_locs = opt.fsolve(self._all_forces_for_balance, ↩
                all_axial_locs)
266         dist = all_axial_locs - soln_locs
267         # Write giant location list back to filaments
268         self.reload_locations(soln_locs)
269         return dist
270
```

```
271     def _all_forces_for_balance(self, locs):
272         """Give axial forces at all points, given all axial ↩
                locations
273 
274         Parameters:
275             locs: Axial locations, given as a large list, ↩
                    containing the
276                  locations of every crown followed by the location ↩
                        of every
277                  actin node. This looks something like:
278                      [thick_0_crown_0, thick_0_crown_1, ..., ↩
                            thick_0_crown_60,
279                      ..., thick_3_crown_60, thin_0_node_0, ..., ↩
                            thin_0_node_89,
280                      ..., thin_7_node_89]
281         Returns:
282             forces: axial force at each of the locations given on ↩
                    entry
283         """
284         # I think that we need to load the locations into the hs
285         self.reload_locations(locs)
286         # Get forces
287         forces = []
288         for thick in self.thick:
289             forces.append(thick.axialforce())
290         for thin in self.thin:
291             forces.append(thin.axialforce())
292         return np.hstack(forces)
293 
294     def flatten_locations(self):
295         """Return a one dimensional version of the current ↩
                locations"""
296         all_axial_locs = []
297         [all_axial_locs.append(thick.axial) for thick in self.thick]
298         [all_axial_locs.append(thin.axial) for thin in self.thin]
299         return np.hstack(all_axial_locs)
300 
301     def reload_locations(self, f_axial):
302         """Pop the flat axial locations back into the half-sarcomere
303 
304         Parameters:
305             f_axial: all axial locations, as a flat list
306         Returns:
307                  nothing
308         """
309         for thick in self.thick:
310             thick.axial = f_axial[0:thick.number_of_crowns]
311             f_axial = np.delete(f_axial, ↩
                    np.s_[0:thick.number_of_crowns])
312         for thin in self.thin:
```

```
313             thin.axial = f_axial[0:thin.number_of_nodes]
314             f_axial = np.delete(f_axial, ↩
                    np.s_[0:thin.number_of_nodes])
315
316     def _get_residual(self):
317         """Get the residual force at every point in the ↩
                half-sarcomere"""
318         thick_f = np.hstack([t.axialforce() for t in self.thick])
319         thin_f = np.hstack([t.axialforce() for t in self.thin])
320         mash = np.hstack([thick_f, thin_f])
321         return mash
322
323     def _get_crossbridges_in_state(self, state):
324         """Return the number of cross-bridge in state x, by face"""
325         assert 0<=state<3, "Valid states are 0, 1, and 2"
326         state_count = []
327         for thick in self.thick:
328             state_count.append([]) # Append list for this thick ↩
                    filament
329             for face in thick.thick_faces:
330                 xb_states = face.get_states()
331                 # Count states that match our passed states of ↩
                        interest
332                 count = sum([state == xb_s for xb_s in xb_states])
333                 state_count[-1].append(count)
334         return state_count
335
336     def _set_timestep(self, timestep):
337         """Set the length of the time step in ms"""
338         [thick._set_timestep(timestep) for thick in self.thick]
339
340     def set_latticespacing(self, ls):
341         """Set the distance between the faces of adjacent ↩
                filaments"""
342         self.lattice_spacing = ls
343
344     def get_frac_in_states(self):
345         """Calculate the fraction of cross-bridges in each state"""
346         nested = [t.get_states() for t in self.thick]
347         xb_states = [xb for fil in nested for face in fil for xb ↩
                in face]
348         num_in_state = [xb_states.count(state) for state in ↩
                range(3)]
349         frac_in_state = [n/float(len(xb_states)) for n in ↩
                num_in_state]
350         return frac_in_state
351
352     def get_crossbridge_properties(self, prop_names):
353         """Create tuples of properties for each cross bridge
354
```

```
355         Parameters:
356             prop_names: list of cross-bridge property names to ↩
                    include
357                 in our resulting output, currently supported are
358                 'numeric_state', 'axial_location', and 'axial_force'
359         Outputs:
360             props: tuple of properties for each cross-bridge
361         """
362         props = []
363         # Go through the property names, retrieving specified ones
364         if 'numeric_state' in prop_names:
365             props.append([xb.get_numeric_state()
366                 for thick in self.thick
367                 for crown in thick.crowns
368                 for xb in crown.crossbridges])
369         if 'axial_location' in prop_names:
370             props.append([xb._get_axial_location()
371                 for thick in self.thick
372                 for crown in thick.crowns
373                 for xb in crown.crossbridges])
374         if 'axial_force' in prop_names:
375             props.append([xb.axialforce()
376                 for thick in self.thick
377                 for crown in thick.crowns
378                 for xb in crown.crossbridges])
379         assert len(props)==len(prop_names), "Oops, didn't get that ↩
                prop name"
380         # Glue the parts together by cross-bridge
381         props = [tuple(p[i] for p in props) for i in ↩
                range(len(props[0]))]
382         return props
383 
384     def get_lattice_spacing(self):
385         """Return the current lattice spacing"""
386         return self.lattice_spacing
387 
388     def get_hiding_line(self):
389         """Return the distance below which actin binding sites are ↩
                hidden"""
390         return self.hiding_line
391 
392     def set_hiding_line(self):
393         """Update the line determining which actin sites are ↩
                unavailable"""
394         farthest_actin = min([min(thin.axial) for thin in ↩
                self.thin])
395         self.hiding_line = -farthest_actin
396 
397     def display_axial_force_end(self):
398         """ Show an end view with axial forces of face pairs
```

```
399
400         Parameters:
401             None
402         Returns:
403             None
404         """
405         # Note: The display requires the form:
406         #  [[M0_A0, M0_A1, ..., M0_A5], ..., [M3_A0, ..., M3_A5]]
407         forces = [[face.axialforce() for face in thick.thick_faces]
408                       for thick in self.thick]
409         # Display the forces
410         self.display_ends(forces, "Axial force of face pairs", True)
411
412     def display_state_end(self, states=[1,2]):
413         """ Show an end view of the current state of the ↩
            cross-bridges
414
415         Parameters:
416             states: List of states to count in the display, defaults
417                     to [1,2] showing the number of bound ↩
                        cross-bridges
418         Returns:
419             None
420         """
421         # Compensate if the passed states aren't iterable
422         try:
423             iter(states)
424         except TypeError:
425             states = [states]
426         # Retrieve and process cross-bridge states
427         # Note: The display requires the form:
428         #  [[M0_A0, M0_A1, ..., M0_A5], ..., [M3_A0, ..., M3_A5]]
429         state_count = []
430         for thick in self.thick:
431             state_count.append([]) # Append list for this thick ↩
                filament
432             for face in thick.thick_faces:
433                 crossbridges = face.get_xb()
434                 # Retrieve states
435                 xb_states = [xb.get_numeric_state() for xb in ↩
                    crossbridges]
436                 # Count states that match our passed states of ↩
                    interest
437                 count = sum([state in states for state in ↩
                    xb_states])
438                 state_count[-1].append(count)
439         # Display the cross-bridge states
440         self.display_ends(state_count, ("Cross-bridge count in ↩
            state(s) "
441                                         + str(states)), False)
```

```
442
443     def display_state_side(self, states=[1,2]):
444         """ Show a side view of the current state of the ↩
                cross-bridges
445
446         Parameters:
447             states: List of states to count in the display, defaults
448                     to [1,2] showing the number of bound ↩
                            cross-bridges
449         Returns:
450             None
451         """
452         # Compensate if the passed states aren't iterable
453         try:
454             iter(states)
455         except TypeError:
456             states = [states]
457         # Retrieve and process cross-bridge states
458         # Note: The display requires the form:
459         # [[A0_0,... A0_N], [M0A0_0,... M0A0_N], ...
460         #  [M0A1_0,... M0A1_N], [A1_0,... A1_N]]
461         instate = lambda x: x in states
462         azo = lambda x: 0 if (x is None) else 1 # Actin limited to ↩
                zero, one
463         oddeven = 0
464         vals = []
465         for thick in self.thick:
466             vals.append([])
467             for face in thick.thick_faces:
468                 m_s = [xb.get_numeric_state() for xb in ↩
                        face.get_xb()]
469                 m_s = map(instate, m_s)
470                 while len(m_s) < 40:
471                      m_s.append(-1)
472                 a_s = [azo(bs.bound_to) for bs in ↩
                        face.thin_face.binding_sites]
473                 if oddeven == 0:
474                      vals[-1].append([])
475                      vals[-1][-1].append(a_s)
476                      vals[-1][-1].append(m_s)
477                      oddeven = 1
478                 elif oddeven == 1:
479                      vals[-1][-1].append(m_s)
480                      vals[-1][-1].append(a_s)
481                      oddeven = 0
482         # Display the cross-bridge states
483         title = ("Cross-bridges in state(s) " + str(states))
484         for fil in vals:
485             for pair in fil:
486                 self.display_side(pair, title=title)
```

```
487
488     def display_ends(self, graph_values, title=None, ↩
            display_as_float=None):
489         """ Show the state of some interaction between the filaments
490
491         Parameters:
492             graph_values: Array of values to display in the format:
493                 [[M0_A0, M0_A1, ..., M0_A5], ..., [M3_A0, ..., ↩
                    M3_A5]]
494             title: Name of what is being shown (optional)
495             display_as_float: Display values as floats? Tries to ↩
                determine
496                 which type of value was passed, but can be ↩
                    manually set to
497                 True or False (optional)
498         Returns:
499             None
500
501         The display is of the format:
502          +-----------------------------------------------------+
503          |           [AA]              [AA]                    |
504          |                                                     |
505          |  [AA]     0200     [AA]     0300     [AA]           |
506          |                                                     |
507          |      0200      0010    0100      0050               |
508          |           (MM)              (MM)                    |
509          |      0100      0010    0100      0010               |
510          |                                                     |
511          |  [AA]     0100     [AA]     0100     [AA]           |
512          |                                                     |
513          |           [AA]     0400     [AA]     0100     [AA]  |
514          |                                                     |
515          |               0200      0020    0200      0020      |
516          |                    (MM)              (MM)           |
517          |               0200      0010    0300      0020      |
518          |                                                     |
519          |           [AA]     0600     [AA]     0300     [AA]  |
520          |                                                     |
521          |                    [AA]              [AA]           |
522          +-----------------------------------------------------+
523         """
524         # Functions for converting numbers to easily displayed ↩
              formats
525         left_float = lambda x: "%-4.1f" % x
526         right_float = lambda x: "%4.1f" % x
527         left_int = lambda x: "%-4i" % x
528         right_int = lambda x: "%4i" % x
529         if display_as_float == True:
530             l = left_float
531             r = right_float
```

```
532         elif type(graph_values[0][0]) == int or display_as_float ↩
                == False:
533             l = left_int
534             r = right_int
535         else:
536             l = left_float
537             r = right_float
538         # Print the title, or not
539         if title is not None:
540             print("  +" + title.center(53,"-") + "+")
541         else:
542             print("  +" + 53*"-" + "+")
543         # Print the rest
544         v = graph_values # Shorthand
545         print(
546         "  |           [AA]               [AA]                ↩
                |\n" +
547         "  |                                                  ↩
                |\n" +
548         "  |  [AA]     %s     [AA]     %s     [AA]           |\n"
549          % (l(v[0][1]), l(v[1][1])) +
550         "  |                                                  ↩
                |\n" +
551         "  |      %s      %s    %s      %s                |\n"
552          % (l(v[0][0]), r(v[0][2]), l(v[1][0]), r(v[1][2])) +
553         "  |           (MM)               (MM)                ↩
                |\n" +
554         "  |      %s      %s    %s      %s                |\n"
555          % (l(v[0][5]), r(v[0][3]), l(v[1][5]), r(v[1][3])) +
556         "  |                                                  ↩
                |\n" +
557         "  |  [AA]     %s     [AA]     %s     [AA]           |\n"
558          % (l(v[0][4]), l(v[1][4])) +
559         "  |                                                  ↩
                |\n" +
560         "  |           [AA]     %s     [AA]     %s     [AA]  |\n"
561          % (l(v[2][1]), l(v[3][1])) +
562         "  |                                                  ↩
                |\n" +
563         "  |               %s      %s    %s      %s       |\n"
564          % (l(v[2][0]), r(v[2][2]), l(v[3][0]), r(v[3][2])) +
565         "  |                  (MM)               (MM)         ↩
                |\n" +
566         "  |               %s      %s    %s      %s       |\n"
567          % (l(v[2][5]), r(v[2][3]), l(v[3][5]), r(v[3][3])) +
568         "  |                                                  ↩
                |\n" +
569         "  |           [AA]     %s     [AA]     %s     [AA]  |\n"
570          % (l(v[2][4]), l(v[3][4])) +
```

```
571         "   |                                                      ↩
               |\n" +
572         "   |                       [AA]                  [AA]          ↩
               |\n" +
573         "   +------------------------------------------------------+")
574         return
575 
576     def display_side(self, graph_values, ends=(0, 0, 0), title=None,
577                      labels=("A ", "M ", "A "), display_zeros=True):
578         """Show the states of the filaments, as seen from their ↩
               sides
579 
580         The input is essentially a list of dictionaries, each of ↩
               which
581         contains the values necessary to produce one of the panels ↩
               this
582         outputs. Each of those dictionaries contains the title (if ↩
               any)
583         for that panel, the side titles, the end values, and the ↩
               numeric
584         interaction values. Currently, the interaction values are ↩
               limited
585         to integers.
586 
587         Parameters:
588             graph_values: Values to display in the format
589                 [[A0_0, A0_1, ..., A0_N],
590                  [M0A0_0, M0A0_1, ..., M0A0_N],
591                  [M0A1_0, M0A1_1, ..., M0A1_N],
592                  [A1_0, A1_1, ..., A1_N]]
593             ends: None or values for ends in the format
594                 [A0_end, M0_end, A1_end]
595             title: None or a title string
596             labels: None or filament labels in the format
597                 ['A0', 'M0', 'A1']
598             display_zeros: Defaults to True
599         Returns:
600             None
601 
602         The printed output is of the format:
603          +-----------------------------------------------------+----+
604          | Z-disk ↩
                                                               | ↩
                   |
605          | ||----*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                        | A0 |
606          | 000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                      |    |
607          | ↩
                                                                     ↩
```

```
          |     |
608     |       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ↩
          000   |     |
609     |       ↩
          #==#==#==#==#==#==#==#==#==#==#==#==#==#======||  ↩
          | M0 |
610     |       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ↩
          M-line |     |
611     | ↩
                                                                  ↩
          |     |
612     | 000   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                  |     |
613     | ||----*--*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                   | A1 |
614     | Z-disk ↩
                                                          | ↩
             |
615     +-----------------------------------------------------------+----+
616     | ↩
                                                                  ↩
          |     |
617     | ||---*--*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                    | A2 |
618     | ↩
                                                                  ↩
          |     |
619     | ↩
                                                                  ↩
          |     |
620     |          ↩
          #==#==#==#==#==#==#==#==#==#==#==#==#==#===||   | ↩
          M1 |
621     | ↩
                                                                  ↩
          |     |
622     | ↩
                                                                  ↩
          |     |
623     | ||---*--*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                    | A3 |
624     | ↩
                                                                  ↩
          |     |
625     +-----------------------------------------------------------+----+
626    ... and so on.
627    """
628    # Functions for converting numbers to easily displayed ↩
          formats
```

```
629         filter_zeros = lambda x: x if (display_zeros or (x != 0)) ↩
               else None
630         l = lambda x: "%-2i" % filter_zeros(int(x))
631         bl = lambda x: "%-3i" % filter_zeros(int(x))
632         r = lambda x: "%2i" % filter_zeros(int(x))
633         br = lambda x: "%3i" % filter_zeros(int(x))
634         # Print the title, if any
635         if title is not None:
636             print("  +" + title.center(134,"-") + "+----+")
637         else:
638             print("  +" + 134*"-" + "+----+")
639         # Print the rest
640         vals = [[bl(ends[0])] + map(l, graph_values[0]),
641                 map(l, graph_values[1]) + [br(ends[1])],
642                 map(l, graph_values[2]),
643                 [bl(ends[2])] + map(l, graph_values[3])] # Shorthand
644         print(
645         "  | Z-disk ↩

               |    |\n" +
646         "  | ↩
               ||----*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--
                                                                    | %s |\n"
647             % labels[0] +
648         "  | %s   %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
                                                                   |    |\n"
649             % tuple(vals[0]) +
650         "  | ↩

               |    |\n" +
651         "  |      %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s    %s  |    |\n"
652             % tuple(vals[1]) +
653         "  |      ↩
               #==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==
                | %s |\n"
654             % labels[1] +
655         "  |      %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s  M-line |    |\n"
656             % tuple(vals[2]) +
657         "  | ↩

               |    |\n" +
658         "  | %s   %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
               %s %s %s %s %s %s %s %s %s %s %s %s %s %s ↩
                                                                   |    |\n"
659             % tuple(vals[3]) +
```

```
660         "   | ↩
                ||----*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--*--
                                                                      | %s |\n"
661              % labels[2] +
662         "   | Z-disk ↩

                |     |\n" +
663         "  ↩
                +--------------------------------------------------------------------
664         )
665         #+-----------------------------------------------------------+----+
666         #| Z-disk ↩
                                                                       | ↩
                   |
667         #| ||----*--*--*--*--*--*--*--*--*--*--*--*--*--... ↩
                   | A0 |
668         #| 000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                       |     |
669         #| ↩
                                                                            ↩
                |     |
670         #|       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                        |     |
671         #|       #==#==#==#==#==#==#==#==#==#==#==#==#==#==#==... ↩
                    | M0 |
672         #|       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                        |     |
673         #| ↩
                                                                            ↩
                |     |
674         #| 000    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                       |     |
675         #| ||----*--*--*--*--*--*--*--*--*--*--*--*--*--... ↩
                   | A1 |
676         #| Z-disk ↩
                                                                       | ↩
                   |
677         #+-----------------------------------------------------------+----+
678         #| ↩
                                                                            ↩
                |     |
679         #|  ...--*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                        | A0 |
680         #|        00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                       |     |
681         #| ↩
                                                                            ↩
                |     |
682         #|       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                        |     |
```

```
683      #|   ...=#==#==#==#==#==#==#==#==#==#==#==#==#==#==#==... ↩
                 | M0 |
684      #|       00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                     |     |
685      #| ↩
                                                                          ↩
            |     |
686      #|         00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ↩
                    |     |
687      #|   ...--*--*--*--*--*--*--*--*--*--*--*--*--*--*--* ↩
                     | A1 |
688      #| ↩
                                                                          ↩
            |     |
689      #+------------------------------------------------------------+----+
690      #| ↩
                                                                          ↩
            |     |
691      #| ↩
                                                                          ↩
            | A0 |
692      #| ↩
                                                                          ↩
            |     |
693      #| ↩
                                                                          ↩
            |     |
694      #|       00 00 00 00 00 00 00 00 00 00   000 ↩
                                |     |
695      #|   ...=#==#==#==#==#==#==#==#==#==#======|| ↩
                              | M0 |
696      #|       00 00 00 00 00 00 00 00 00 00   M-line ↩
                             |     |
697      #| ↩
                                                                          ↩
            |     |
698      #| ↩
                                                                          ↩
            |     |
699      #| ↩
                                                                          ↩
            | A1 |
700      #| ↩
                                                                          ↩
            |     |
701      #+------------------------------------------------------------+----+
702      #
703      #
704      return
705
```

```
706
707 sarc = hs()
708 #class hsTests(unittest.TestCase):
709 #    def setUp(self):
710 #        pass
711 #
712 #
713 #if __name__ == '__main__':
714 #    unittest.main()
```

B.5 File run.py

```
 1 #!/usr/bin/env python
 2 # encoding: utf-8
 3 """
 4 run.py - control a run on an aws node
 5 
 6 Created by Dave Williams on 2010-11-19
 7 """
 8 
 9 import sys
10 import os
11 import time
12 import uuid
13 import cPickle as pickle
14 import shelve
15 import copy
16 import optparse
17 import multiprocessing as mp
18 import boto
19 import hs
20 import numpy.random as random
21 
22 ## Settings
23 BUCKET_NAME = 'model_results' # For results files
24 FOLDER_NAME = 'test_disregard/' # Needs trailing /
25 
26 
27 ## Help message
28 help_mesg = "Around here, we like to call our functions with a ↩
       little \n \
29         thing called options. Here's how we'd go about it: \n \
30         usage: %prog [options] arg1 arg2 "
31 
32 ## Logging
33 def log_it(message):
34     """Print message to sys.stdout"""
35     sys.stdout.write("run.py " + mp.current_process().name +
36             " ## " + message + "\n")
```

```
37      sys.stdout.flush()
38
39  ## Run status
40  def run_status(i, total_steps, sec_left, sec_passed, process_name):
41      if i%50==0 or i==0:
42          sec_left=int(sec_left)
43          sys.stdout.write("\n %s has finished %i/%i steps, ↩
               %ih%im%is left to go"%(process_name, i+1, total_steps, ↩
               sec_left/60/60, sec_left/60%60, sec_left%60))
44          sys.stdout.flush()
45
46  ## Push file to S3
47  def push_to_s3(s3conn, file_name,
48                 bucket_name=BUCKET_NAME,
49                 folder_name=FOLDER_NAME):
50      ## Fix S3 folder name if needed
51      if folder_name[-1] != '/':
52          folder_name += '/'
53      ## Connect to bucket on S3
54      try:
55          bucket = s3conn.get_bucket(bucket_name)
56      except boto.exception.S3ResponseError:
57          log_it('Bucket connection gives error, trying to create ↩
               bucket')
58          bucket = s3conn.create_bucket(bucket_name)
59      ## Upload the file
60      #os.system('s3cmd put %s s3://%s/%s/%s'%(file_name, bucket_name,
61      #                                        folder_name,
62      #                                        ↩
           file_name.split('/')[-1]))
63      key_name = folder_name+file_name.split('/')[-1]
64      key = bucket.new_key(key_name)
65      key.set_contents_from_filename(file_name)
66      key.close()
67
68  ## Data creation functions, the run and the recording callback
69  def run_callback(sarc, input={}):
70      """Choose what to log, return it"""
71      atpase = lambda s: ↩
           sum(sum(s.last_transitions,[]),[]).count('31')
72      appendd = lambda n,v: input[n].append(v)
73      appendd('ax',     sarc.axialforce())
74      appendd('ra',     sarc.radialforce())
75      appendd('rt',     sarc.radialtension())
76      appendd('fb',     sarc.get_frac_in_states())
77      appendd('atpase', atpase(sarc))
78      return input
79
80  def run(sarc, timesteps, queue=None, folder_name=FOLDER_NAME,
81          local_path=os.path.expanduser('~/data/')):
```

```
82      """Create data, save it locally, upload it to S3 and log to ↩
            SDB"""
83      random.seed() #Must do in process to get proper dice rolls
84      ## Create data names and open files
85      result_name = local_path + time.strftime('%Y%m%d.') + ↩
            str(uuid.uuid1())
86      meta_name = result_name+'.meta.pkl'
87      data_name = result_name+'.data.pkl'
88      sarc_name = result_name+'.sarc.shelf'
89      if os.path.exists(local_path) is False:
90          os.makedirs(local_path)
91      meta_file = open(meta_name, 'w')
92      data_file = open(data_name, 'w')
93      sarc_file = shelve.open(sarc_name, protocol=2)
94      ## Make the data and log it
95      results = {'ax':[], 'ra':[], 'rt':[], 'fb':[], 'atpase':[]}
96      tic = time.time()
97      for tstep in range(timesteps):
98          sarc.timestep()
99          results = run_callback(sarc, results)
100         sarc_file[str(tstep)]=copy.deepcopy(sarc)
101         # Update on how it is going
102         toc = int((time.time()-tic) / (tstep+1) * ↩
                (timesteps-tstep-1))
103         run_status(tstep, timesteps, toc, time.time()-tic,
104                    mp.current_process().name)
105     ## Write to disk
106     # Close and compress the large sarcomere copy file
107     sarc_file.close()
108     try:
109         os.system('7za a -mx=3 %s %s'%(sarc_name+'.7z', sarc_name))
110         sarc_name += '.7z'
111     except:
112         print "Couldn't compress, is 7zip installed?"
113     # Saving smaller datas
114     results['sarc'] = sarc #Nice to have one copy in your hatband
115     pickle.dump(results, data_file, 2)
116     data_file.close()
117     # Save metadatas
118     meta_data = {'folder':folder_name,
119                  'name':result_name,
120                  'run_length': timesteps,
121                  'lattice_spacing': sarc.lattice_spacing,
122                  'z_line': sarc.z_line}
123     pickle.dump(meta_data, meta_file, 0)
124     meta_file.close()
125     ## Pass the data and meta filenames to the queue
126     if queue is not None:
127         queue.put(data_name)
128         queue.put(meta_name)
```

```
129         queue.put(sarc_name)
130     return (meta_name, data_name, sarc_name)
131
132 ## Just like calling the main
133 def like_main(ls=14, sl=1250, ts=10, ub=FOLDER_NAME, multiproc=1, ↩
        loops=1):
134     return main(['--lattice_spacing', str(ls),
135                  '--z_line', str(sl),
136                  '--timesteps', str(ts),
137                  '--folder', str(ub),
138                  '--multiprocessing', str(multiproc),
139                  '--loops', str(loops)])
140
141 ## Our main man
142 def main(argv=None):
143     ## Get our args from the command line if not passed directly
144     if argv is None:
145         argv = sys.argv[1:]
146     ## Parse arguments into values
147     print argv
148     parser = optparse.OptionParser(help_mesg)
149     parser.add_option('-t', '--timesteps', dest="timesteps",
150                       default=10, type='int',
151                       help='number of timesteps for each run [10]')
152     parser.add_option('-l', '--loops', dest="loops",
153                       default=1, type='int',
154                       help = 'how many runs to do on each core [1]')
155     parser.add_option('-s', '--lattice_spacing', dest="ls",
156                       default=None, type='float',
157                       help='lattice spacing to a number [14.0]')
158     parser.add_option('-z', '--z_line', dest="z_line",
159                       default=None, type='int',
160                       help='z-line where the thin filaments end ↩
                            [1250]')
161     parser.add_option('-b', '--bucket', dest="bucket_name",
162                       default=BUCKET_NAME, type='string',
163                       help='set the bucket to which results will ↩
                           be uploaded')
164     parser.add_option('-f', '--folder', dest="folder_name",
165                       default=FOLDER_NAME, type='string',
166                       help='set the folder to which results will ↩
                           be uploaded')
167     parser.add_option('-m', '--multiprocessing', ↩
            action="store_const",
168                       dest="proc_num", default=1, ↩
                            const=mp.cpu_count(),
169                       help='run as many copies as there are cores ↩
                            [False]')
170     parser.add_option('--halt', action="store_true",
171                       dest="the_end_of_the_end", default=False,
```

```
172                         help='shutdown computer on completion ↩
                               [False]')
173     (options, args) = parser.parse_args(argv)
174     ## For each loop we are supposed to run
175     for loop in range(options.loops):
176         ## Initialize the half-sarcomeres with passed ls and z-lines
177         sarcs = [hs.hs(options.ls, options.z_line) for i in
178                   range(options.proc_num)]
179         ## Set up queue
180         queue = mp.Queue()
181         ## Set up processes
182         process = [mp.Process(target=run, args=(s, ↩
               options.timesteps,
183                              queue, options.folder_name)) for s in sarcs]
184         ## Start processes, wait for them to finish
185         [p.start() for p in process]
186         [p.join() for p in process]
187         ## Upload the results
188         log_it("Uploading results of this loop")
189         s3conn = boto.connect_s3()
190         while not queue.empty():
191             push_to_s3(s3conn, queue.get(),
192                          options.bucket_name,
193                          options.folder_name)
194         log_it("Gonna run runs "+str(options.loops-1-loop)+" more ↩
               times")
195     if options.the_end_of_the_end is True:
196         os.system("sudo shutdown now -h")
197     return 0 #Successful termination
198
199
200 if __name__ == "__main__":
201     sys.exit(main())
```

Appendix C

APPENDIX C

This code launches an on-demand cluster of computers on Amazons Elastic Compute Cloud. It was used to perform the simulations which generated the data used in Chapters 3 and 4. The created cluster listens for job parameters loaded into a selectable Simple Queue Service First-In-First-Out queue.

C.1 File aws.py

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3  """
4  easy_cluster.py - Make a simple cluster that listens to SQS
5
6  Created by Dave Williams on 2011-11-07
7  """
8
9  import os
10 import sys
11 import time
12 import copy
13 import optparse
14 import boto
15
16 ## Defaults
17 BASE_PATH = os.path.expanduser('~/code/multifil/')
18 CODE_LOCATION = BASE_PATH + 'hs'
19 CODE_BUCKET = 'model_code'
20 USER_DATA = BASE_PATH + 'aws/sqs_control_userdata_script.py'
21 KEY_FILE = os.path.expanduser('~/.aws/keys/controller_keypair.pem')
22 KEY_NAME = 'controller_keypair'
23 SECURITY_GROUP = 'default'
24 AMI32BIT = ('ami-d812efb1', 'c1.medium') # Debian 6.0 derived
25 AMI64BIT = ('ami-c40df0ad', 'c1.xlarge') # Debian 6.0 derived
26 USE_SPOTS = True
27 SPOT_BIDS = (0.15, 0.60) #In cents for the 32 and 64 bit types
28 NUM = 2 # Number of instances to launch
29 BIGNUM = 20 # Number of instances to launch in a big cluster
```

```
30
31  S3 = boto.connect_s3()
32  EC2 = boto.connect_ec2()
33
34  def quick_print(string):
35      """Print the given string straight to the stdout"""
36      sys.stdout.truncate(0)
37      sys.stdout.write(string)
38      sys.stdout.flush()
39      return
40
41  def load_code_to_s3(code_loc = CODE_LOCATION, bucket_name = ↩
        CODE_BUCKET):
42      """Load .py files from the local code tree to the S3 code ↩
            bucket"""
43      bucket = S3.get_bucket(bucket_name)
44      for path, dirs, files in os.walk(code_loc):
45          for filename in files:
46              if filename.endswith('.py'):
47                  key = ↩
                        bucket.new_key(path[len(code_loc):]+'/'+filename)
48                  key.set_contents_from_filename(path+'/'+filename)
49
50  def start_spot_instances(num = NUM, ami = AMI32BIT, bid = ↩
        SPOT_BIDS[0],
51                           key_name = KEY_NAME, sec=SECURITY_GROUP,
52                           user_data_filename = USER_DATA):
53      """Start a number of spot instances and return them as a list"""
54      if os.path.getsize(user_data_filename) > 16*1024:
55          print "error: User data file is too big"
56          return
57      reservation = EC2.request_spot_instances(
58          price           = bid,
59          image_id        = ami[0],
60          count           = num,
61          key_name        = key_name,
62          security_groups = [sec],
63          user_data       = open(user_data_filename, ↩
                'r').read(16*1024),
64          instance_type   = ami[1])
65      time.sleep(.5) # Give the machines time to register
66      #nodes = copy.copy(reservation[0].instances)
67      #return nodes
68      return reservation
69
70  def start_on_demand_instances(num = NUM, ami = AMI32BIT,
71                                key_name = KEY_NAME, ↩
                                      sec=SECURITY_GROUP,
72                                user_data_filename = USER_DATA):
```

```
73      """Start a number of on-demand instances and return them as a ↩
            list"""
74      if os.path.getsize(user_data_filename) > 16*1024:
75          print "error: User data file is too big"
76          return
77      reservation = EC2.run_instances(
78          image_id        = ami[0],
79          min_count       = num,
80          key_name        = key_name,
81          security_groups = [sec],
82          user_data       = open(user_data_filename, ↩
                'r').read(16*1024),
83          instance_type   = ami[1])
84      time.sleep(.5) # Give the machines time to register
85      nodes = copy.copy(reservation.instances)
86      return nodes
87
88  def launch_cluster(num = NUM, ami = AMI32BIT, spot_instances = ↩
        USE_SPOTS,
89                     bid = SPOT_BIDS[0]):
90      """Prep and launch a cluster"""
91      quick_print("Uploading code to S3 \n")
92      load_code_to_s3()
93      quick_print("Creating reservation \n")
94      if spot_instances == True:
95          nodes = start_spot_instances(num, ami, bid)
96          ids = [node.id for node in nodes]
97          node_states = lambda nodes: [node.state == 'active'
98                                       for node in nodes]
99          node_update = lambda : ↩
                EC2.get_all_spot_instance_requests(ids)
100     else:
101         nodes = start_on_demand_instances(num, ami)
102         ids = [node.id for node in nodes]
103         node_states = lambda nodes: [node.state_code == 16 for ↩
                node in nodes]
104         node_update = lambda : [inst for res in ↩
                ec2.get_all_instances(ids)
105                                 for inst in res.instances]
106     # Waiting until all nodes are ready
107     quick_print("Nodes are starting \n")
108     while not all(node_states(nodes)):
109         nodes = node_update()
110         ready = sum(node_states(nodes))
111         quick_print("\r%i of %i nodes are ready"%(ready, ↩
                len(nodes)))
112         time.sleep(1)
113     quick_print("\nAll nodes ready \n")
114     return nodes
115
```

```
116 def big_cluster_launch(num = BIGNUM, ami = AMI64BIT,
117                        spot_instances = USE_SPOTS,
118                        bid = SPOT_BIDS[1]):
119     """A bigger version of launch_cluster"""
120     return launch_cluster(num, ami, spot_instances, bid)
121 
122 def kill_cluster(cluster):
123     """Terminate the cluster nodes"""
124     try:
125         [node.stop() for node in cluster]
126     except:
127         [node.cancel() for node in cluster]
128         ids = [node.instance_id for node in cluster]
129         [instance.terminate() for reservation in ↩
            EC2.get_all_instances(ids)
130          for instance in reservation.instances]
131 
132 def main(argv=None):
133     """Called when run as a script from the command line"""
134     ## Get our args from the command line if not passed directly
135     if argv is None:
136         argv = sys.argv[1:]
137     ## Parse arguments into values
138     parser = optparse.OptionParser()
139     parser.add_option('-n', '--number', dest='num', default=None,
140                       type='int', help='Number of nodes to launch')
141     parser.add_option('-a', '--ami', dest='ami', ↩
        default=AMI64BIT[0],
142                       type='str', help='AMI ID to launch')
143     parser.add_option('-t', '--type', dest='type', ↩
        default=AMI32BIT[1],
144                       type='str', help='Type of instance to start')
145     parser.add_option('-b', '--big', dest='big', default=False,
146                       action='store_true', help='Launch a big ↩
                          cluster')
147     parser.add_option('-d', '--ondemand', dest='spot', default=True,
148                       action='store_false', help='Use on-demand ↩
                          instances')
149     parser.add_option('-p', '--bidprice', dest='bid', ↩
        default=SPOT_BIDS[0],
150                       type='float', help='Price to bid on spot ↩
                          instances')
151     (options, args) = parser.parse_args(argv)
152     if options.big is True:
153         bid = SPOT_BIDS[1] if options.bid == SPOT_BIDS[0] else ↩
            options.bid
154         nodes = big_cluster_launch(
155             num = options.num if options.num!=None else BIGNUM,
156             spot_instances = options.spot,
157             bid = bid)
```

```
158     else:
159         nodes = launch_cluster(
160             num = options.num if options.num!=None else NUM,
161             ami = (options.ami, options.type),
162             spot_instances = options.spot,
163             bid = options.bid)
164     return nodes
165
166 if __name__ == "__main__":
167     sys.exit(main())
```

VITA

Dave Williams grew up in Yakima, a smallish city located within spitting distance of Washington State's geographic center. He spent his childhood playing in the desert, enjoying desserts, and electrocuting the occasional pickle. He attended Reed College as an undergraduate, where he learned the the mysteries of nuclear reactor operation and the heart from Ariah Kidder. After completing his BA in physics he decided, "Why not add a 'bio' on there?" and joined the department of Physiology and Biophysics at the University of Washington, intending to study motor proteins. In the Daniel and Regnier labs, he enjoyed prodding computers into decent muscle-simulation machines and prodding Tom into producing bad puns. He and Ariah, now married, continue to love travel and will be moving to Boston to find out more about this "East Coast" thing they keep hearing about. There Dave will poke at pigeons with Andy Biewener. To this day, he doesn't care for pickles.

He is likely to be found at `http://www.charlesdavidwilliams.com`.