THE ADAPTIVE MULTISCALE SIMULATION INFRASTRUCTURE

William Reading Tobin

Submitted in Partial Fulfillment of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

Approved by:
Dr. Mark S Shephard, Chair
Dr. Catalin Picu
Dr. Christopher Carothers
Dr. Barbara Cutler

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York

[August 2018]
Submitted July 2018

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I would first like to thank my adviser, Prof. Mark S. Shephard. His guidance was invaluable and provided me a path to navigate the many obstacles may otherwise have knocked me off course throughout my graduate career.

I would like to the other members of my committee, Prof. Catalin Picu, Prof. Christopher Carothers, and Prof. Barbara Cutler both for serving on my committee, and for the education they have provided me during my tenure at RPI.

I would like to thank my peers in the Scientific Computation Research Center and associated laboratories and groups including Cameron Smith, Bryan Granzow, Dan Ibanez, Victor Chan, Daniel Fovargue, Seegyoung Seol, Max Bloomfield, Jacob Merson, and Sai Deogekar. I've learned so much from you all and I look forward to continuing to learn and grow as a scientist for many years.

I owe my graduate career and what is yet to come entirely to my undergraduate advisers Prof. Scott Burgess and Prof. Christopher Dugaw at Humboldt State University (HSU). Prof. Burgess was instrumental in teaching me the fundamentals of Computer Science and helped immensely in understanding the hierarchy of abstractions that provide a cohesive model of computation. Prof. Dugaw helped me realize my skill and abilities in mathematics and provided me the push I needed to pursue a dual major during my undergraduate studies. Both were incredibly supportive of my educational interests and goals and instrumental in my pursuit of higher education.

Further I would like to thank my friends and colleagues from my time at RPI: Michael Caiola, Timothy Krentz, Christopher Bertucci, Taylor Dorsey, Ted Frater, Dan Eckhart, Jennifer Crodelle, Joe Crodelle, Jordan Angel, Brandon Bensel, Savannah Dorsey, Kris Zieb, Wunyin Au, Peter Muller, and many others. My time here with you all was some of the best of my life and I will always cherish it, thank you all for your unflagging friendship. I would further like to thank my oldest, best friend and brother by choice Patrick T. Colley for the last 15 years of camaraderie and commiseration.

Finally I would like to thank my immediate family: William M., Jani, Aubree, Billy, and Caden. I love you all, thank you for all the love and support you gave me over the literal lifetime it took to reach this point: I would not have made it this far without you.

ABSTRACT

Increases in the computational power of massively parallel machines have facilitated the implementation and execution of numerical simulations of unprecedented scale and fidelity. When the physical mechanism governing the action of interest in a simulation occurs at a scale orders removed from the desired resolution of the solution, even modern leadership class machines are insufficient to provide full granularity. Multi-scale numerical simulations allow the coupling of multiple scales of interest in order for the multi-scale simulation to reflect important properties of each scale.

The whole-clothe implementation of multi-scale simulations targeting massively parallel machines requires a substantial investment of man-hours. Leveraging existing, well-established single-scale simulations already in wide use in order to construct new multi-scale systems will allow a reduction in development time and costs to produce novel new multi-scale systems.

The design and development of the Adaptive Multi-scale Simulation Infrastructure (AMSI) along with the design and implementation of a multi-scale soft biological tissues code is discussed. The parallel characteristics and implications of the AMSI component-code coupling approach are considered in the context of the soft tissue multi-scale code.

Modern simulations on leadership-class machines also leverage dynamic and adaptive capabilities to load balance simulations during execution and reduce time-to-solution. Integration of such codes into a multi-scale system must allow for the use of dynamic load balancing operations while maintaining all inter-scale linkages.

The deployment of codes designed for massively parallel machines relies on the leveraging of various component libraries, often fulfilling similar purposes but targeted at specific machine architectures to provide optimal performance. Accessing these component libraries through mechanisms providing for the use of a generic interface over a component code targeted to a particular platform allows for the development of algorithms once and their deployment and optimization on various platforms without the need to explicitly modify code. Implementation of such a mechanism should necessarily introduce minimal overhead into the resulting simulation, with a goal of zero-overhead abstraction of the various component backends.

CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Introduction

Multi-scale numerical simulations have undergone rapid advancement in recent years. Increases in the computational power of massively parallel machines have facilitated the implementation and execution of numerical simulations of unprecedented scale and fidelity. However, when the physical mechanism governing the action of interest in a simulation occurs at a scale orders removed from the desired resolution of the solution, even modern leadership class machines are insufficient to simulate the domain with full granularity.

No fully general model or methodology for the development and implementation of multi-scale simulations yet exists. Many approaches have been developed and efforts are ongoing to define a general multi-scale modeling ontology and implementation formalisms to produce and execute new simulation codes. Beyond the development and implementation phases of a general multi-scale simulation production methodology there are numerous challenges with respect to the rapidly advancing standards of the HPC space.

Thousands of man-hours have been devoted to the development of effective simulation codes and software suites. This existing set of proven simulation capability provides a rich source of development components in the construction of new multi-scale simulations [1].

The method of implementation for most massively parallel numerical simulation codes is single-program multiple-data (SPMD), where each processing core associated with the simulation is assigned some subset of the data to process, taking part in parallel communication (in-memory or message passing) where necessary to continue valid computation. This implementation paradigm is versatile and widely used, and exhibits good parallel scaling characteristics when leveraged correctly.

The bespoke development of multi-scale simulations using the SPMD paradigm necessarily allows for the highest level of granularity with respect to implementation restrictions specific to the HPC space, incorporating many specific optimization strategies. The use of the SPMD strategy to produce a combined, tightly-coupled multi-scale simulation code necessarily requires the extensive modification of existing codebases or the redundant development of capabilities already provided in the set of single-scale simulation components.

Developing capabilities to incorporate existing SPMD codes to produce novel multi-scale systems avoids repetitious developments and will allow for a reduction in initial development times [2].

The effective deployment of a simulation code on a variety of hetero- and homo-geneous computing platforms presents an additional consideration in the development of methods for code implementation.

Hardware developments in recent years have resulted in HPC machines with less architectural homogeneity. Many machines on the current Top500 list include computational accelerators such as the Xeon Phi and NVIDIA K40. The introduction of the accelerator into the HPC space has complicated the implementation of new codes in order to leverage the full power offered by such devices. Even with such complications, the SPMD paradigm remains the dominant implementation strategy for massively parallel simulation code development.

The operations developed to support code development on a variety of HPC machines must result in adequate performance, utilization, and parallel scaling metrics in the resulting simulation codes. The varying restrictions placed on the development of such a multi-scale coupling technique to combine existing predominantly SPMD simulation codes in a massively parallel distributed execution environment in a manner generally applicable to all classes of multi-scale simulation represents a substantial set of difficulties.

Toward the production of a general methodology of multi-scale software implementation we developed the Adaptive Multi-scale Simulation Infrastructure (AMSI), a set of libraries and tools to support the implementation of multi-scale simulations targeted at HPC machines, and supporting parallel dynamic and adaptive simulation techniques.

1.2 Outline

In order to make the construction of multi-scale simulation codes accessible to developers in the HPC space, we have develop a set of libraries supporting the general operations required to couple together existing single-scale simulation codes. The AMSI `multiscale` library provides the computational abstractions and operations required to construct a new multi-scale simulation with minimal intervention in the existing operations of a single-scales simulation code. We discuss the implementation of a multi-scale soft biological tissue simulation using the capabilities provide by AMSI `multiscale`. We extend the capabilities of the AMSI `multiscale` library to allow for the dynamic migration of single-scale data

in the context of a coupled multi-scale simulation and use these capabilities to implement a dynamic load balancing operation on the multi-scale soft tissue simulation.

Further we provide a method of implementing zero-overhead abstraction layers for use in implementing algorithms for deployment on machines with varied architectural considerations.

Chapter 2 introduces key features of multi-scale simulation methodologies and discusses existing multi-scale frameworks developed to support multi-scale simulation. Chapter 2 also discusses the design and development of the Adaptive Multi-scale Simulation Infrastructure (AMSI), a set of libraries supporting the implementation of a multi-scale simulations through the combination of existing single-scale simulation codes. Further it provides a discussion of the implications of key design decisions on the parallel structure and performance of a resulting application code.

In chapter 3 the development of a multi-scale application for the simulation of soft biological tissues using the AMSI libraries discussed in chapter 2 is covered. An overview of the mechanics being simulated at each scale in the simulation is provided, and a description of the scale-linking mechanisms is discussed, as well as the implementation of the scale-coupling procedure using the AMSI libraries. Several examples of test problems constructed and run with the multi-scale system are provided, as well as the results of these runs. The parallel properties of the simulation are also explored with discussion of how they related to the design of the AMSI multi-scale functionality.

Chapter 4 provides a review of the concepts of dynamic load balancing in the context of HPC applications. Further it discusses load balancing individual simulation scales in the context of a multi-scale application developed using AMSI, including additional requirements placed on the load balancing procedure and an algorithmic description of the process of scale-sensitive load balancing required to retain inter-scale validity in multi-scale simulations. Usage of these load balancing operations on the multi-scale soft tissue simulation from chapter 3 is discussed.

In chapter 5 the development of abstraction layers for use in HPC applications is discussed. Goals and requirements for such APIs is provided, and a method of implementing such abstraction layers in C++ is developed. This approach is used to implement an abstraction layer for interacting with linear algebraic data structures (LAS). The LAS abstraction layer is used in the implementation of the soft tissue code from chapter 3, and for several key

operations in the under-development multi-scale fusion codes XCGM and M3D-C1. Comparison between use of the provided optimized abstraction layer versus several other forms of commonly-used abstraction layer implementation and direct usage of a third party backend are provided.

Finally in chapter 6 directions for extension and future work are discussed in more depth, with brief road maps and considerations for continued development in these areas.

1.3 Contributions

This thesis primarily concerns the implementation of multi-scale simulations on HPC machines. During the work performed to create this thesis, novel contributions have been achieved through:

- Developing and implementing a set of software libraries to ease the implementation of new multi-scale simulations and interactions through the combination of existing single-scale simulations.
- Development of a multi-scale simulation for modeling soft tissues by combining independent single-scale simulation codes.
- Applying the multi-scale soft tissue simulation code to problems of interest in the scientific community today.
- Allowing the use of existing load-balancing techniques for improving computational performance of individual scale-codes in a multi-scale simulation while retaining inter-scale linkages.
- Making use of the new load-balancing capabilities in the soft-tissue simulation to attempt to increase simulation performance.
- The design and implementation of abstraction layers with zero overhead, possessing properties allowing both for the ease of swapping out underlying libraries and supporting usage in HPC environments.
- Application of the LAS zero-overhead abstraction layer for linear algebraic operations in multiple codebases.

CHAPTER 2
THE ADAPTIVE MULTI-SCALE SIMULATION INFRASTRUCTURE – AMSI

2.1 Multi-scale Simulation

The development of new numerical simulations has proliferated rapidly in the last several decades. The dramatic increases in computational power of massively parallel machines has facilitated the implementation and execution of numerical simulations of unprecedented scale and fidelity. When the physical mechanism governing the evolution of the system occurs primarily at a scale orders removed from the desired resolution of the solution, even massive increases in computation power is insufficient in many cases. Multi-scale simulation techniques offer a method of executing simulations and producing results on a domain orders of magnitude removed from key underlying phenomena originating from finer scales and accounted for through the application of scale-coupling methods.

Multi-scale simulation broadly falls into two categories: concurrent and hierarchical problems. These categories are most readily distinguished by the relationships between the interacting scales in a problem.

In concurrent problems the domain of each scale is modeled using similar physical dimensions: the standard units used to describe the scale will typically fall within several orders of magnitude of each other. Interface surfaces or overlap regions are used to transfer relevant physical tensor field information between scales. The transfer of relevant tensor field values allow the scales to influence one another and gives rise to multi-scale behaviors in the problem simulation.

The Domain Decomposition Methods (DDM) [3] and the Quasicontinuum Methods (QM) [4] both allow the implementation of concurrent multi-scale problems. DDM allows the implementation of multi-physics continuum-to-continuum coupling while QM is used to couple atomistic models to continuum regions in a macro-scale domain.

Hierarchical problems are characterized by strongly-separated scale behaviors. The domain relationship between two such scales is often reduced to the existence of "micro-scale" simulations occurring at various points throughout the larger-scale simulation. The

tensor field couplings which give rise to multi-scale behavior occur at each point in the macro-scale simulation at which a micro-scale simulation is embedded. The local value of tensor field data is used to establish conditions at the micro-scale, which in turn produces a tensor field value relevant to the evolution of the macro-scale simulation. Hierarchical multi-scale simulations typically require up- and down-scaling operations to account for differences between the coupled scales.

Hierarchical problems can be implemented with the use of information passing schemes in which a well-defined set of parameters is transmitted between the simulation scales [5]–[7]. This approach effectively takes out the use of a constitutive model to guide the evolution of a simulation and replaces it with the response of a set of coupled mirco-scale simulations placed at distinct points through the macro-scale domain.

The Heterogeneous Multiscale Methods (HMM) [8], [9] are methods used to solve multi-scale problems with either localized regions of macroscopic model failure or problems without a known macroscopic model entirely. HMM is well-suited for application to hierarchical problems as it does not require the existence of a macro-scale constitutive model, allowing the micro-scale simulations to fully guide system behavior.

An example of scale separation maps [10] associated with these multi-scale problem categories can be seen in figure 2.1. Further discussion of the taxonomy of multi-scale modeling as given in [11] is a useful primer on the various categories of multi-scale models, with accompanying examples for each category to further elucidate the concepts. Implementation of multi-scale numerical simulations typically rely on either an ad-hoc set of algorithms designed for a particular problem or on implementation within a framework. Ad-hoc approaches, by their nature, are targeted to a particular multi-scale problem for a specific set of specifically-coupled single-scale models, and thus offer little or no opportunity to effectively implement alternative multi-scale simulation capabilities. Framework approaches are more flexible and admit a broader range of reuse, but typically necessitate that single-scale operations be adapted to work in the ecosystem they create.

Many thousands of man-hours have been devoted to the development of various industry standard single-scale simulation codes and suites. The development of tightly-coupled multi-scale codes wherein the individual scales are not separate at the implementation level is possible, but such an approach would often involve redundant development of capabilities already available in existing – highly developed – physics codes. Developing capabilities to

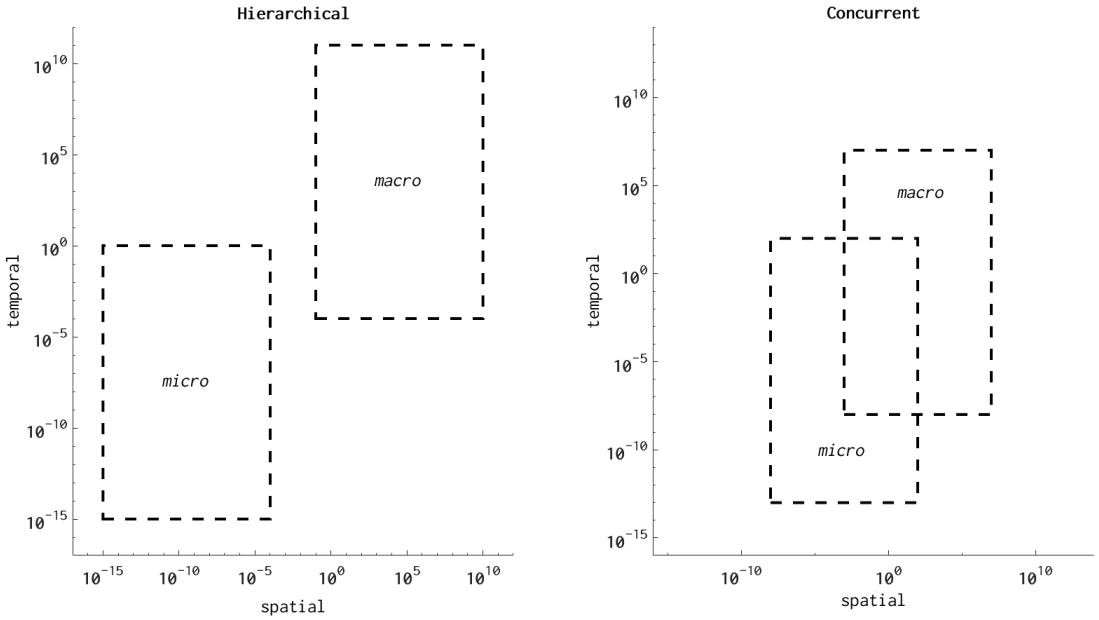

Fig. 2.1: Hierarchical and Concurrent Scale Separation Maps

incorporate existing codes to produce novel multi-scale systems avoids repetitious developments and will allow for a reduction in initial development times [2]. This minimization of imposed requirements on single-scale codes being coupled – accompanying the requirement that any HPC code should exhibit good parallel performance in a variety of metrics including scaling and process utilization – forms the basis of the requirements for our multi-scale system.

2.2 Multi-scale Frameworks

Foremost a multi-scale framework needs to support the introduction and solution of each of the single-scale models involved in the multi-scale model. It must provide the capability to link introduced scales together through the communication of data between physical tensor fields existing at each scale. Finally it must accomplish the above goals while executing effectively on large scale parallel computing systems required to provide the level of computation needed for such problems.

Most generally, software intended to support multi-scale simulations needs to "enable the introduction of new models, algorithms, and data structures" [12], to effectively incorporate new single-scale models and incorporate them into novel multi-scale systems.

Multi-scale frameworks should minimize the required development time in order to

allow developers to quickly iterate on new multi-scale simulation designs as they develop new models. As such, the refactoring of existing codes (single-scale simulation codes) a developer may want to use as a basis for the development of new multi-scale simulations should be minimized.

The public interfaces used to by an application developer to implement new multi-scale coupling of distinct scales should be understandable and use generic types and hooks where appropriate in order to allow a developer to supply new data structure and extend the behavior of coupling algorithms.

A set of well-defined modules with distinct interfaces should be implemented, especially as this concerns data ownership and simulation state. State data should be query-able through interfaces in order for coupled codes to maintain consistency throughout the simulation. Implementing general code, or providing mechanisms to extract important underlying data in general formats is necessary.

With these qualities relating to extensability, multi-scale application developers would be required to adhere to specific data structures and algorithms, or write their own wrapper code in order to translate specialized between specialized data structures. These issues and rationales concerning multi-scale software development are further discussed in [12].

Beyond extensability the simulation code produced through use of a framework must be performant. A primary focus on HPC machines is parallel scalability of the resulting code in order to apply the code to problems of sufficient granularity to provide meaningful insight. Beyond parallel scaling of the simulation, the resulting code should also maximize process utilization in order to (1) reduce time-to-solution and (2) to use the computational resources available as cost-effectively as possible.

The extensability and performance focuses often conflict in practice, as extensible and generic development priorities may sacrifice performance and the most performant code usually sacrifices ease of extensability. There are many different approaches taken toward balancing these issues. Some frameworks choose to focus primarily on producing the most performant code possible, which typically requires a developer extensively refactor any existing codebase in order to make use of the framework features and achieve the goal of performant code. Conversely some prefer to allow for easy extensability to allow quick development turnaround so the exploration of novel multi-scale models can occur more rapidly.

Given the extensive existing single-scale codebases in use in both industry and academia

and the countless man-hours which have gone into their development, our approach is to focus primarily on ease of extensability to allow the incorporation of existing, legacy codebases into new multi-scale simulations. Performance is considered and prioritized as well, but in cases where a method of increasing performance would sacrifice simplicity of implementation priority is given to ease of implementation.

There are numerous multi-scale frameworks currently in development intended to facilitate the implementation of multi-scale simulations on HPC machines. We do not intend to provide a full survey of such frameworks in this section but rather to provide representative examples that take alternative approaches as to be able to see the strengths and weaknesses of these approaches.

The Uintah Problem Solving Environment (PSE) [13] is a framework developed at the University of Utah intended to allow the intuitive combination of parallel simulation components into novel scientific applications, including but not limited to multi-scale simulations. Each component in the Uintah system is designed to solve a PDE on a structured AMR grid, and must adhere to a C++ API used to establish connections between components. All components are decomposed into a set of discrete tasks, each of which has a well-defined set of required input and output data. The information about these tasks is expressed in the form of a directed acyclic graph (DAG) [14], and is used by the Uintah system to schedule task execution across the HPC system. The parallel communication necessary to provide task input and output occurs through a data warehousing structure which controls access to local and global data.

The Mesh-Based Parallel Code Coupling Interface (MpCCI) [15] is a library which facilitates the coupling of simulation codes based on the underlying domain grid over which simulation data (specifically tensor field values) are distributed. Support for different grid formats is provided by MpCCI, as well as different methods of data interpolation when needed to communicate data between simulation scales with mismatched grids. MpCCI uses a code-coupler component object in order for two single-scale application codes to communicate with eachother [16]. A coupler object is implemented to transform values from a producing single-scale code into values that can be used by a consuming single-scale code.

In the time since MpCCI was developed over a decade ago it has seen sufficent adoption to warrant the organization of several user forums and is currently used to support the implementation of multi-physics codes. There have also been coupler objects implemented

to allow many industry-standard simulation tools to be coupled through MpCCI including Abaqus, ANSYS Fluent, MATLAB, and others [17].

The MUSCLE-HPC [18] code is a partial implementation of the MUSCLE-2 toolkit [19], [20] for use on HPC machines. The multi-scale simulation in MUSCLE-HPC is constructed from a set of submodels, each of which is implemented by inheriting from an abstract C++ class which provides an API for initialization, boundary condition updates, numerical solution iteration, and several data query functions. Coupling together of submodels into a multi-scale model is done through the use of an interface of Conduits which the class API has methods to express the set of input and output Conduits for the submodel. Further the MUSCLE-HPC runtime environment requires that a machine core acting as a server is devoted to running a special Manager class to register client submodels and facilitate the creation of parallel communication channels between the submodels.

The Multiphysics Object Oriented Simulation Environment, MOOSE [21], [22] is a multiphysics code developed to support the implementation of simulations in nuclear engineering. MOOSE is an end-to-end framework in that it attempts to incorporate both the required physical kernels, the discrete mesh and tensor fields, and the linear solver components. It makes use of a third party linear solver in PETSc [23]–[25] and third party nonlinear solvers in Trilinos NOX [26]–[31]. However it is not intended to supply a component model to allow incorporation of user components into new multi-scale simulations, but rather to construct multi-scale simulations from a suite of in-house components.

2.3 AMSI

The Adaptive Multi-scale Simulation Infrastructure (AMSI) [32] is a set of libraries providing support for the implementation and execution of dynamic multi-scale simulations on massively-parallel HPC machines. AMSI is designed to ease the incorporation of legacy single-scale simulation codes into new multi-scale simulations. These codes are typically implemented using a single-program multiple-data (SPDM) parallel implementation strategy. Using these component codes in a multi-scale infrastructure which is also SPMD in nature would require extensive development to accomodate, either on behalf the AMSI user, or with respect to the structure of AMSI itself. In contrast taking a multiple-program multiple-data (MPMD) parallelization strategy with AMSI allows the inclusion of multiple existing component codes without the requirement they are extensively reworked or accommodated

in the AMSI systems. The MPMD model has been noted to be more appropriate for multi-physics and multi-scale systems [33].

An AMSI user can integrate their single-scale codes using AMSI without an imposed requirement that they refactor their data structures or interfaces in order to accommodate the AMSI runtime. The more monolithic frameworks such as UINTAH, `MOOSE`, and MpCCI require that a user adapt their code explicitly to interfaces specified by the framework system. AMSI attempts to take a service library approach by providing new capabilities to existing codes rather than a framework approach where the entire simulation construction and execution procedure is controlled using the framework.

In the minimal case the work to adapt a codebase involves simply the implementation of wrapper code or intermediate interface layers (such as the Coupler components used in MpCCI), but in the worst case this requires a restructuring of much of the codebase to accommodate requirements of the framework. The `MUSCLE2` framework requires that integrated sub-models expose a specific interface so the coupling daemon is able to query them about required inputs and output. In UINTAH all user-code must be expressed in a specialized asynchronous task format. MpCCI requires the implementation of code-coupler components to facilitate the coupling of codes with specific domain formats. The `MOOSE` system is and end-to-end multiphysics system to inclusion of external component codes isn't considered a primary goal.

The single-scale codes used in AMSI have no requirements placed on them with respect to a specific underlying format of spatial discretization or tensor field data structures. In contrast codes being coupled using the MpCCI framework must adapt their domain querying interfaces to facilitate the necessary interrogation operations for relating the geometric domains [15]. The Uintah Software explicitly manages the discretized domain of the problem – implemented using a adaptive structured grid-type model – and user-developed simulation components operate on this domain managed by the system [34]. The `MOOSE` multiphysics system also explicitly manages the domain discretization using structured grids.

AMSI does not explicitly require domain interrogation functionality to implement multi-scale coupling operations, this allows for the integration of code with arbitrary (or nonexistent) discrete domain representations. This protects the capabilities provided by AMSI against a shift in community adoption of new simulation models or different domain representations.

AMSI requires placement of multi-scale planning and coupling calls only at those locations in legacy codebases associated with the actual coupling operations used for multi-scale interaction. All AMSI interfaces operate on built-in data types or are implemented to accept generic data types, limiting the need to refactor existing codebases to adapt to specialized data types or inheritance schemes [35].

There are many open questions in the development and verification of multi-scale modeling approaches [36]. AMSI attempts to provide the tools to approach these questions through the development of new multi-scale models and simulations from existing single-scale simulations without placing assumptions on the multi-scale model or simulation implementation that would preclude exploration of an open question.

2.4 Model Design

The design of the simulation coupling operations utilizing the AMSI multiscale library is informed by two abstract models. The first is the Abstract Scale Model (ASM), which describes a generalized single-scale simulation. The second is the Abstract Coupling Model (ACM), which describes a generic coupling of two such simulations. These models provide a workflow through which a simulation developer can produce a new multi-scale simulation on an HPC machine using the facilities provided by the AMSI libraries. Further, these models guided many development decisions during the implementation of the AMSI multiscale library.

The theoretical model-production tool discussed in [37] consists of a set of decisions made about various features of a multi-scale model and where each feature lies within the taxonomy introduced and discussed in that document. The abstract models used in the development of both AMSI and particular multi-scale simulations using AMSI provides sufficient information for the unique specification of each decision in the theoretical tool discussed, and thus allows a multi-scale model produced by AMSI to be defined in terms of the discussed multi-scale taxonomy.

In addition to taxonomic considerations, the AMSI model workflow includes considerations for the hierarchy of implementation operations that go into the production of a new multi-scale code.

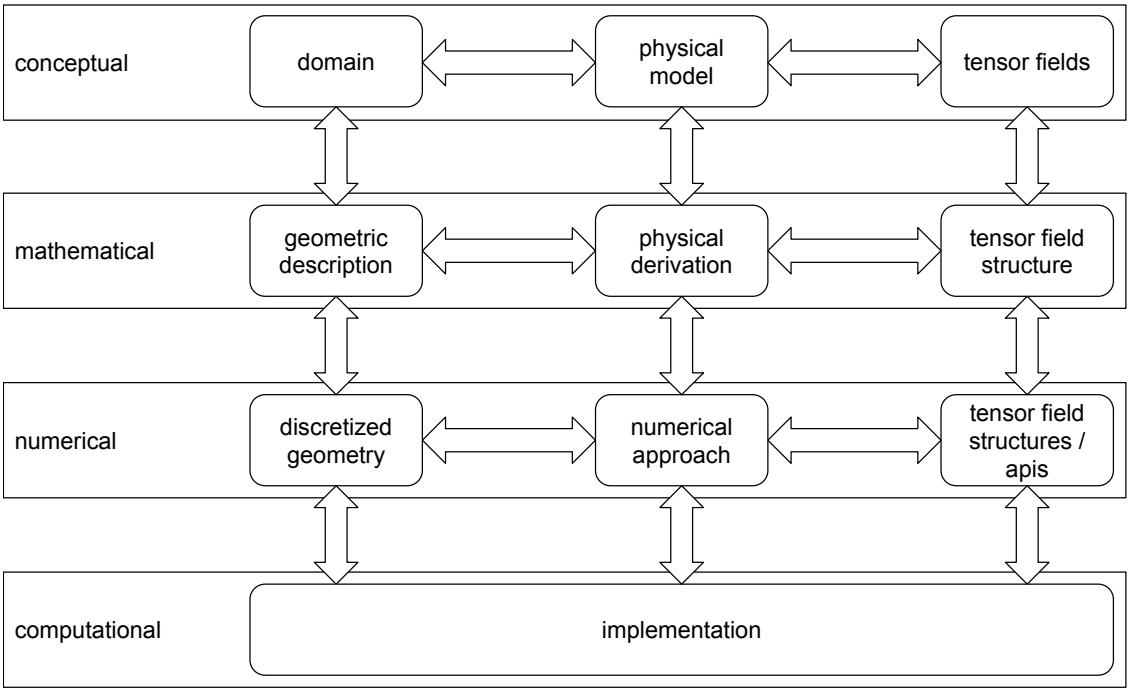

Fig. 2.2: Abstract Single-scale Model

2.4.1 Abstract Scale Model (ASM)

The Abstract Scale Model (ASM) (illustrated in figure 2.2) provides a hierarchical model of the implementation specifics of a single-scale simulation. The model is oriented on the principle of most general abstraction through the steps required to produce the actual computational model used along the vertical axis. Each level in the abstraction hierarchy requires information in three different areas: the physics being simulated, the domain (type) over which the simulation is to take place, and the physical tensor fields required to quantify the specific problem. By specifying or implementing the information required at each node in the ASM a developer will fully describe and produce a single-scale simulation.

The conceptual physical model is often the starting point in the development of an ASM, as it informs many of the decisions made during the specification of the rest of the model. This high-level description is often specified in terms of the primary physical property(ies) being modeled, e.g. displacements, temperature, trajectories, etc, as well as (typically) one or more conservative laws. The specification of the physical model most directly effects the conceptual domain and conceptual tensor fields that are required. The conceptual domain incorporates both the temporal and spatial domains of the problem, which provides sufficient information to plot the scale on a scale-separation and interaction graph. In addi-

tion the conceptual domain and physical model combined provide sufficient information to specify the qualities the domain must posses in order to support the modeled mechanics, e.g. continuum domain, atomistic domain. The conceptual tensor fields are directly influenced by the physical model and includes every primary and supporting field required to simulate the physical model as specified by the chosen conservative laws.

The mathematical physical derivation is a specification of the chosen form of the conservative law(s) from the conceptual physical model and accompanying mathematical requirements. The mathematical geometric description of the domain incorporates the specification of the method of mathematical representation of a domain for the problem, e.g. for a continuum problem a CAD format or grid structure [38], [39]. The mathematical tensor field structure for the required fields is specified, including symmetries which allow for tensor order reductions to simplify the mathematical formulations in the physical derivation.

The numerical approach includes the approach to solving the mathematical physical derivation this includes the construction and solution of the discretized model. The discretized geometry specifies the representation of the geometric domain the numerical approach will query and operate on, this includes for instance meshes and grids for continuum problems and discrete particles for MD problems. The tensor field structures and APIs are specified over the discretized geometry and are queried for use in the numerical approach to produce a solution which is also expressed in terms of a tensor field structure.

The computational implementation of an ASM requires the combination of all numerical specifications in order to produce a final simulation.

2.4.2 Abstract Coupling Model (ACM)

The Abstract Coupling Model (ACM) (seen in figure 2.3) provides a model of the coupling of two distinct realizations of an ASM (fig 2.2). The hierarchy of each coupled ASM must be taken into account during the specification of the primary nodes in the ACM. It is convenient to view the ACM as a top-down view of two ASMs and the coupling layer between them. Each of the conceptual, mathematical, and numerical layers must still be considered when specifying nodes in the ACM. Specifying and implementing the ACM fully will result in a valid multi-scale coupling operating between two single-scale simulations. The ACM is derived from a model initially discussed in [1].

The physical model compatibility requires the specification of the relationship between

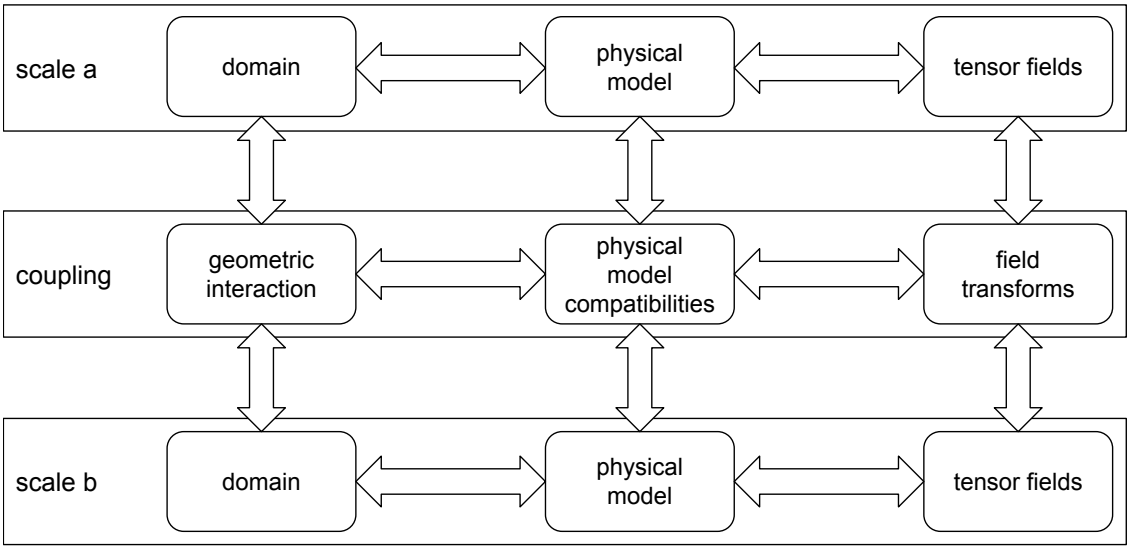

Fig. 2.3: Abstract Multi-scale Coupling Model

the two physical models. This includes any laws relating the physical quantities modeled in the two ASMs, and may require intermediate operations including derived values in order for the two physical models to influence each other. Proceeding down the ASM hierarchy this further requires a particular mathematical derivation, as well as the specification of any numerical operations required to produce all quantities needed to satisfy the mathematical derivation.

The geometric interaction of problem domains includes the specification of each domain on a scale-separation map, as well as the nature of the domain relationship with respect to e.g. overlapping regions/interfaces or full locality of sub-domains for strongly-separated scales. This specifies whether the multi-scale problem is concurrent or hierarchic as discussed in sections 2.1. Going down the hierarchy this also includes any mathematical operations required to map spatially between the domains which can include e.g. inverting mappings, interpolation operations, tensor field queries at selected locations. At the numerical level the operations necessitated by the mathematical relationships must be implemented.

The field transforms in a coupling require the specification of all operations required for influences from a field associated with a scale to be incorporated into a field on the other scale(s) in the ACM. At the conceptual level this includes the full set of tensor fields on each scale along with known relevant relationships between the fields. The specification of these operations is primarily mathematical and numerical in nature, and is influenced heavily by the physical model compatibility and geometric interactions.

2.5 Parallel Design

Software intended to suppport multi-physics simulations needs to "enable the introduction of new models, algorithms, and data structures" [12]. Further, a set of well-defined modules with distinct interfaces must be implemented, especially as this concerns data ownership and simulation state. State data must be query-able through interfaces in order for coupled codes to maintain consistency throughout the simulation. Further, implementing general code, or providing mechanisms to extract important underlying data in general formats is necessary. This forces multi-physics application developers to adhere to specific data structures and algorithms, or write their own wrapper code in order to translate specialized data structures into more general formats. These issues and rationales for multi-scale software development are further discussed in [12].

Toward this end AMSI provides a simple API used for multi-scale coupling and scale-sensitive load-balancing operations, and makes no requirement on the application code providing any interfaces for AMSI usage. This allows intervention in legacy code only in locations where multi-scale operations are to be implemented, and where replacement of global collective operations with scale-collective operations is required in order to prevent parallel deadlock.

The design and operation of the AMSI libraries and the multi-scale simulations created through AMSI coupling operations must account for the parallel design of the individual scale codes being coupled. Legacy single-scale simulation codes operate under the assumption that they have the entirety of the global parallel execution space within which to operate. The parallel execution space is the set of threads of execution in a given simulation and their relationship with the physical hardware of the machine (generally analogous to global parallel communicator). A great deal of effort is expended during the development of such single-scale codes into ensuring that as the size of the parallel allocation for the code grows, the parallel operations do not inhibit the computational efficiency of the code.

Allowing two or more single-scale codes designed around the assumption of global parallel execution to interleave their operation in a general manner would be fraught with design and execution pitfalls. Further, the resulting code would likely have few or no guarantees with respect to acceptable parallel performance characteristics: not only in terms of scaling but also process utilization and idle time overheads.

Because of this restriction AMSI avoids the issue of shared access to the global parallel

execution space and instead isolates each scale-code in disjoint subsets of the global parallel communicator. This partitioning of the parallel execution space allows the allocation for each scale-code to be determined to match the computational demand associated with the multi-scale system and/or the current specific problem.

This hierarchy of parallel execution spaces gives rise to a hierarchy of parallel operations on those spaces. The hierarchy of parallel operations in the AMSI system is global, inter-scale, and intra-scale.

Global parallel operations are still possible in an AMSI multi-scale simulation, but AMSI itself does not implement any fully-global parallel operations beyond the capability to partition the global parallel space. Efficient use of any global parallel communication operations (whether synchronous or asynchronous) in a multi-scale system is exceedingly difficult, thus in order to discourage the practice AMSI provides no direct support for such operations.

Inter-scale operations are parallel over the union of the two parallel spaces taking part in the operation, and are designed to be one-way. These operations are referred to as Reconciliations within AMSI, as this type of operation is most often used to reconcile the current simulation state (or metastate) between two disjoint scale-codes. Reconciliations are used to plan and execute coupling operations in order to provide data produced at one scale to another scale for use in continuing the simulation.

Intra-scale operations are parallel over a single parallel execution space in a multi-scale system. These operations are referred to as Assemblies within AMSI, as this type of operation is most often used to produce a final valid product concerning the local simulation state (or, again, metastate) on a specific simulation scale. Assemblies are often used just prior to reconciliations to produce a data structure that is complete over an entire scale, for use in the reconciliation operation. Specifically, when planning coupling operations, the metastate concerning the distribution of discrete coupling values across a scale is Assembled just prior to the Reconciliation operation to produce a coupling plan between the scales (see section 3.2.1 for an example of coupling planning).

2.6 AMSI Parallel Implementation

The AMSI multiscale library provides the facilities to instantiate, manage, and execute dynamic scale coupling operations on otherwise uncoupled single-scale simulation codes.

AMSI specifically targets the use-case of a developer taking existing single-scale simulation codes and combining them to produce a new multi-scale code while minimizing intervention in the legacy codebases.

The core of the AMSI multiscale systems operates at the numerical and implementation levels of the ASM and ACM abstraction hierarchy. While developers use a top-down approach to specify and construct a simulation, initial development on AMSI multiscale has focused on providing functionality in a bottom-up fashion. The intent of this approach is to provide a useful set of functionality for developers before working up the abstraction hierarchy to provide facilities for operations required at those levels.

At the lowest-level, the AMSI multiscale library is concerned with the instantiation and management of the individual single-scale codes used in a multi-scale simulation. The parallel structure of HPC machines requires that in order to couple multiple single-scale codes into a multi-scale code, the relationship of those individual single-scale codes to the machine must be modeled by AMSI multiscale.

AMSI models the parallel execution space of the whole multi-scale simulation as well as each single-scale code in the multi-scale system, through the use of a Scale.

2.6.1 AMSI Scales

The Scale is a data structure modeling the relationship between the computational level of a single ASM used in the construction of a multi-scale simulation and the parallel execution space of the HPC machine the simulation is executing on. The relationship between the Scale and the parallel execution space is specified by the user using an AMSI multiscale configuration file, further discussed in section 2.6. All Scales present in a multi-scale simulation are instantiated and configured at the beginning of a multi-scale simulation.

Several methods for determining the partitioning of the parallel execution space and assigning the partitions to Scales in a simulation are provided. Scale allocations may be determined by total process count, process ratios between multiple Scales, and additionally may take advantage of machine introspection operations such as using the `hwloc` [40] hardware locality library in order to take advantage of machine structure for parallel communication operations.

Also during initialization, a main-like function is associated with an individual Scale. A developer may simply reuse the main function of the existing single-scale simulation, however

the function is additionally passed an `MPI_COMM` associated with the Scale on which parallel operations for the single-scale code modeled by the Scale are to take place. See listing 2.1 for the Scale creation and management API. Currently the initialization of Scales can occur through use of configuration files (discussed in 2.6.2) or the API. The scale main-like function must be set programmatically using the `setExecFunction()` on the scale.

Listing 2.1: AMSI Scale API

```
1  // Main-like execution function taking standard argc and argv
2  //  arguments in addition to an MPI_Comm for configuration
3  typedef int(*exe_func)(int&,char**&,MPI_Comm);
4
5  // create a scale with size-many processes with rank pids[0-size-1]
6  //  pids are specified in the AMSI_COMM_WORLD
7  Scale * createScale(int size, int * pids, const char * nm);
8  // retrieve a scale that has already been created,
9  //  use to retrieve non-local scales when using file-based
10 //  multi-scale configuration
11 Scale * retrieveByName(Scale * scl, const char * nm);
12 // retrieve the currently active local scale
13 // NOTE: currently only a single scale may be active
14 //       per-process
15 Scale * getLocalScale();
16 void destroyScale(Scale * scale);
17
18 // set the main-like function for this scale to execute
19 void setExecFunction(Scale * scale, exe_func e);
20 // execute the main-like function for this scale if
21 //  this scale has an execution function and the
22 //  local rank is assigned to the scale
23 int execute(Scale * scale);
24
25 // whether the local rank is assigned to the scale
26 bool assignedTo(Scale * scale);
27 // translate a pid from a rank in the MPI_Comm associated
28 //  with the scale to a pid in AMSI_COMM_WORLD
29 int localToGlobalRank(Scale * scale, int lcl);
30 // get the MPI_Comm associated with this scale
```

```
31 MPI_Comm scaleComm(Scale * scale);
```

2.6.2 AMSI Scale Instantiation

AMSI Scales are created during simulation initialization, prior to the main-like functions of the scales being called. The Scales can be created programatically using the interface in listing 2.1, or they can by configured using an AMSI configuration file as seen in listing 2.3. When the `initMultiscale()` function (from listing 2.2 is called using the standard `argc` and `argv` arguments from the simulation main, the configuration file – passed in the with the `-a` command-line flag – is parsed and the portions relevant to multi-scale initialization are used to create the specified Scales. The `initAnalysis()` function in listing 2.2 is also called and supplied the AMSI configuration file. It parses the sections specific to initializing third-part analysis packages and initializes them as needed. The analysis and multiscale AMSI packages are entirely orthogonal, so a user can make use of the multiscale coupling package without being concerned with the analysis libraries.

Listing 2.2: AMSI initialization API

```
1  // pass in a configuration file using command-line
2  //  flag -a [filename] to automatically allocate processes
3  //  and create named scales
4  void initMultiscale(int argc,
5                      char * argv[],
6                      MPI_Comm cm = MPI_COMM_WORLD);
7  void freeMultiscale();
8
9  // pass in a configuration file using the command-line
10 //  flag -a [filename] to automatically initialize specified
11 //  analysis libraries
12 void initAnalysis(int argc,
13                   char * argv[],
14                   MPI_Comm cm = MPI_COMM_WORLD);
15 void freeAnalysis();
```

Listing 2.3 shows the full configuration file used for a run of the soft tissue multiscale simulation discussed in chapter 3. In this case the simulation is being assigned to 256

processes, 16 of which are assigned to the micro-scale and 240 of which are assigned to the macro-scale, as can be seen in the `@scales` section. Further the `@relations` section specifies that the macro-scale is Coupled to (sending coupling data to) the micro-scale, and the micro-scale is Coupled to the macro-scale. As will be discussed in section 2.6.3, Couplings are not assumed to be symmetrical.

Listing 2.3: Multiscale Configuration file

```
1  @scales
2  macro 16
3  micro 240
4  @relations
5  macro micro
6  micro macro
7  @util
8  results $RESULTS_DIR
9  @analysis
10 petsc true
11 petsc_options petsc_opts
12 simmetrix true
13 simmetrix_license license.txt
```

2.6.3 AMSI Couplings

The Coupling is a data structure modeling the relationship of units of coupling data between two distinct Scales. The Coupling in AMSI multiscale is not assumed to be symmetric so in order to establish a two-way coupling between Scales a Coupling must be produced for each direction of communication. While it is often the case that a coupling is bi-directional, this is not always the case [37], so this form is preferred. Each Coupling exists on the union of the parallel execution spaces associated with both Scales being coupled, and specifies on Scale as the producer and the other as the consumer.

The Coupling operates at the computational level of the ACM, but is directly informed by specifications at the numerical level. Individual units of scale-coupling data are supplied to the Coupling, e.g. tensor field values located at points as specified by the domain relationship in the ACM. These may be raw values or values derived locally, i.e. the tensor field transformation operations may by applied to the coupling data on either end of the

Coupling. This allows for minimization of communication overhead whenever possible.

A unit of coupling data generically refers to any data type as the precise implementation or data structure of the data is not assumed by the Coupling or AMSI multiscale until the communication phase of the coupling operation.

The parallel discretization of the numerical domains of the multi-scale problem informs the parallel locality of both the origin and the destination of each unit of scale-coupling data. At present this mapping is taken care of by the developer. Facilitating the automation of the construction of this mapping is the next step in advancing the development of AMSI multiscale up the ASM/ACM abstraction hierarchy.

Listing 2.4: AMSI Coupling and Coupling Data API

```
1  // declare coupling data on a scale, returns NULL if
2  //  the local rank is not associated with the scale
3  CouplingData * createCouplingData(Scale * scale, const char * nm);
4  void destroyCouplingData(CouplingData * dat);
5
6  bool assembleCouplingData(CouplingData * dat);
7
8  typedef Coupling * (*coupling_algo)(CouplingData * dat,
9                                      Scale * frm,
10                                     Scale * to);
11
12 Coupling * evenlyDistributedCoupling(CouplingData * dat,
13                                       Scale * frm,
14                                       Scale * to);
15 Coupling * createCoupling(CouplingData * dat,
16                           Scale * frm,
17                           Scale * to,
18                           coupling_algo algo =
19                             evenlyDistributedCoupling);
20 Coupling * invertCoupling(Coupling * orig);
21 // create an empty coupling to reconcile coupling planning
22 //  information into when recv-ing
23 Coupling * createRecvCoupling(Scale * frm, Scale * to);
24 void destroyCoupling(Coupling * coupling);
25
```

```
26 bool assembleCoupling(Coupling * coupling);
27 bool reconcileCoupling(Coupling * coupling);
28
29 Scale * getSender(Coupling * coupling);
30 Scale * getRecver(Coupling * coupling);
```

2.7 AMSI Performance Measures

AMSI is designed around the composition of existing single-scale codes which are most often implemented in a single-program multiple-data (SPMD) style in order to gain performance from parallelization. Despite sharing a common implementation pattern, each single-scale code in a multi-scale system has distinct scaling characteristics that contribute to the scaling characteristics of the multi-scale system. These scaling characteristics are primarily determined with respect to the problem size being worked on, under some metric of computational demand. Two well-established scaling measures are the strong scaling and weak scaling; in strong scaling the problem size is held constant globally while the parallel execution space grows, in weak scaling the problem size is held constant per-process in the execution space as the execution space is increased.

The parallel execution of a problem in an AMSI multi-scale simulation must take into consideration the global problem size and the relationship between the problem sizes for each coupled scale pair. To determine the parallel partitioning and scale-assignment for a particular multi-scale composition and global problem size, the 'performance ratio' of the coupled scales is used, in conjunction with the actual optimization goal.

The simplest optimization goal for simulations is time-to-solution, which for simulations using SPMD parallelization is often attained by simply increasing the parallel execution space, scaling characteristics permitting. The second most common optimization goal is process utilization, which is simply the ratio of useful compute time to total simulation time, thus any process idle time is penalized. Improving process utilization often improves time-to-solution, as a given problem size represents a constant amount of computational effort to be expended, thus less idle time means expending the required computational effort more quickly.

An AMSI multi-scale simulation houses individual scales in partitions of the global execution space, there is an amount of idle overhead inherent in this multi-scale composition

approach. Individual scales may be optimized with respect to process utilization, but there is typically some amount of time where parts of (or an entire) scale must block and wait for the result of some coupled scale. Individual scale optimization in terms of process utilization is often accomplished through load balancing operations. Multi-scale load balancing introduces additional constraints and operations to the load balancing process, which are discussed in detail in 4.

Strong and weak-scaling metrics are still applicable to AMSI multi-scale simulations, but the strong and weak scaling of a particular simulation is additionally a property of the parallel partitioning of the execution space. Thus the partitioning ratio between each pair of coupled scales is a heavily contributing factor to the scaling characteristics exhibited by a multi-scale MPMD simulation. In the simplest multi-scale case of only two coupled scales, this causes the scaling curves to become scaling surfaces due to the introduction of a single coupling-ratio parameter. Introducing additional coupled scales into a simulation also adds additional ratio parameters. The case of three mutually-coupled scales is of particular interest from a performance optimization standpoint, but has not been explored yet and is an area of interest for future work.

Discussion of scaling studies conducted on a multi-scale code implemented using AMSI can be found in section 3.5.

2.8 Conclusion

In this work we have developed an abstract model of single-scale simulation and a multi-scale simulation coupling that is compatible with existing multi-scale modeling approaches. We have developed this model and defined the process through which the model is made concrete in order to implement a multi-scale simulation. To realize these efforts we have implemented the Adaptive Multi-scale Simulation Infrastructure to facilitate the implementation of multi-scale simulations, providing especially for the combination of existing single-scale simulation codes into new, novel multi-scale codes without undue development overhead. We have used the abstract scale model and abstract scale coupling model in conjunction with considerations for the established properties of existing codes in the HPC space to develop a library supporting the flexible creation of multi-scale systems from concrete Scale objects and Coupling objects. Finally we have established that the multi-scale simulation codes resulting from using the AMSI facilities for simulation implementation

exhibit non-typical parallel behavior and explored the implications and evaluation of such codes.

CHAPTER 3
IMPLEMENTATION AND EVALUATION OF A HIERARCHICAL MULTI-SCALE SOFT TISSUE SIMULATION USING AMSI

3.1 Soft Tissue Simulation

The Biotissue code [41] is a multi-scale simulation for modeling soft organic tissues, consisting of an engineering-scale simulation using the finite element method, and a micro-scale quasistatics force-balance simulation based on networks of 1D structural entities. These two codes are combined into a multi-scale code using the coupling features of the AMSI libraries.

The biotissue simulation falls under the HMM classification of numerical approaches. See [42] for additional discussion of the current biotissue simulation and applying the simulation to a problem modeling individual neurons embedded in collagen gels.

Stress and stress-derivative values needed to compute the elemental tangent stiffness matrices and force vectors are supplied by the micro-scale force-balance simulations, which occur at each numerical integration point in the engineering-scale mesh. Appendices A and B contain derivation and implementation details for the macro- and micro-scale problems and appendix C discusses the derivation of the multi-scale coupling terms.

3.1.1 Engineering Scale

The macro-scale geometric domain is described by a parasolid CAD model derived from one of several sources. Some domains are constructed from voxel data recovered from segmented images of actual biological cells/structures, such as the neuron domain in section 3.4.3 and the podocyte domain in section 3.4.2. Other, simpler domains can be constructed as CAD models using geometric modeling software such as the tensile test 'dogbone' in section 3.4.1.

The Simmetrix [43] software suite is used to develop many of the models (particularly from segmented image data), and to produce discretized meshes from the geometric models. All extant domains have been discretized into simulation meshes using tetrahedral elements,

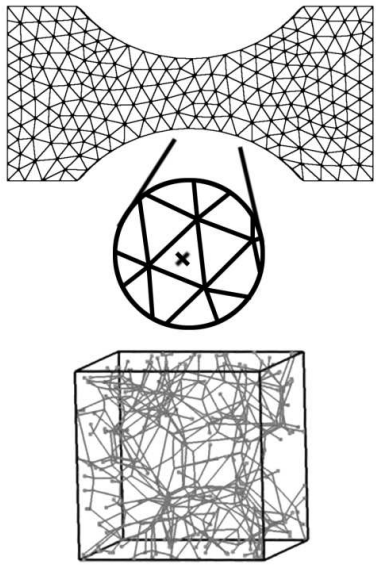

Fig. 3.1: Biotissue Multi-scale Domain Hierarchy

although the simulation makes no strict restriction on the element type or field order used.

Analysis cases are specified on the geometric model (again using the Simmetrix software suite), by tagging geometric entities with information about material properties, micro-scale relationships and structures to load, dirichlet and neumann boundary conditions, and solution strategies including convergence approaches and thresholds and psuedo-timestep specification.

A vector field using first order Langrange shape functions over each element is defined and used in elemental numerical integration routines to construct elemental linear systems for assembly in the global tangent stiffness matrix and load vector. The use of first order Lagrange shape functions induces the creation of a single numerical integration point inside each element in the macro-scale mesh. Each of these numerical integration points may be coupled to a micro-scale representative volume element (RVE) simulation which provides quantities for use in the elemental integration procedures. Whether an element is coupled to an RVE, and the precise RVE domain (or library of RVE domains) to which an integration point is related may be specified on a per-geometric region basis in the Simmetrix simmodeler analysis case specification.

3.1.2 Micro Scale

Each RVE is a dimensionless unit cube containing a connected collagen fiber network, currently all fiber members are truss elements and the fiber network is generated using Delauney triangulation. This results in a high coordination number at each node and consequently a numerically stable global Jacobian given the use of truss elements. Using a Voronoi map to generate the fiber-network structure is also possible, though would also necessitate the use of beam elements for the fiber networks in order to achieve a more stable Jacobian.

The macro-scale incremental displacements $u^{(M)}$ generated by the accumulation of solutions $\Delta u^{(M)}$ from previous simulation psuedo-timesteps and defined on each engineering-scale finite element node are used to calculate the deformation gradient F at each point associated with a micro-scale simulation. The deformation gradient is used to displace the corners of the dimensionless RVE. The fiber-network nodes located on the boundary faces of the RVE are displaced using linear interpolation of the RVE corners displacements and held fixed.

To generate an initial guess $\Delta u_0^{(m)}$ for the micro-scale Newton-Raphson procedure the interior fiber-network nodes are displaced using trilinear interpolation of the corner displacements in the first macro-scale iteration of each pseudo-time step. In all subsequent micro-scale iterations the solution from the previous iteration is used to provide a first order approximation of the displacement.

After each micro-scale simulation converges to a solution, the Jacobian and force vector are then formulated again using the solved fiber network state. The force exerted along the boundary of the RVE is summed for each of the principal and shear components of a stress tensor, which is then dimensionalized and sent back to the engineering-scale for use in the elemental tangent stiffness matrix formulation.

More detail into the derivation of this multi-scale system can be found in [44]–[47]. Discussion of the dimensionalization of the dimensionless force terms produced by each fiber network can be found in [48], [49].

The multi-scale simulation consists of two nested Newton-Raphson operations occurring. During each macro-scale Newton iteration every micro-scale RVE must undergo a full Newton-Raphson convergence to produce coupling data for the macro-scale. As such, the micro-scale nonlinear processes will typically dominate the total computational effort required to fully converge to a solution at each macro-scale psuedo-timestep.

3.2 Soft Tissue Implementation and Multi-scale Coupling with AMSI

A configuration file is used to initialize AMSI, partition the parallel execution space, and pass the partitioned spaces to the single-scale codes for use in their individual initialization procedures. After initialization, the macro-scale code creates the necessary data structure describing the distribution of coupling data (in this case the deformation gradient at every numerical integration point in the discrete domain) across the macro-scale parallel space. This data structure is supplied to a Coupling which is Reconciled with a null Coupling object present on the micro-scale.

As the scales are strongly-separated, and the multi-scale interaction is information-passing rather than concurrent, no geometric information must be dealt with prior to constructing the Coupling object used to couple macro-scale to micro-scale. Further, each micro-scale simulation is simple enough (approx. 10k unknowns), that they execute in serial rather than parallel, as the parallel communication overhead involved in solving a step in a single micro-scale RVE would increase overall time to solution in the average case. This allows the macro-scale to construct a Coupling object without any coordination with the micro-scale simulation. One of the built-in information-passing Coupling construction algorithms supplied with AMSI is sufficient for this purpose, and produces a Coupling evenly distributing the coupling terms across the micro-scale simulation.

After an initial guess generated by a linear-elastic constitutive model is generated, the micro-scale is sent initial information including fiber-parameters, macro-scale element type, element coordinates, and the first deformation gradient. This data is applied to the micro-scale simulations to bring their state into agreement with the current macro-scale state.

Once the micro-scale receives the initial data an induced Coupling is created, which in turn creates an induced data structure describing the allocation of the macro-scale coupling terms across the micro-scale space, which coincides with the distribution of micro-scale coupling terms to be sent back to macro-scale.

Since the allocation of RVEs to micro-scale processes is handled by the macro-scale, the macro-to-micro Coupling can simply be inverted to produce the micro-to-macro coupling, as the macro-scale processes which require micro-scale data are precisely those which sent out the data in the macro-to-micro coupling operation. After the micro-scale simulation converges, the forces along the surface of the RVE are gathered and used in the micro-to-

macro coupling operation.

3.2.1 Scale Implementations

Both the macro and micro-scale simulations are implemented using a variety of simulation tools and libraries. As indicated below, several in-house SCOREC libraries are used (primarily related to mesh and tensor field interactions) and several third party libraries are used as well (primarily for linear algebraic solvers, some geometric model/mesh operations, and analysis attribute specification).

3.2.1.1 Engineering Scale Implementation

The macro-scale geometry and discrete meshing are handled by Simmtrix [43] as noted in 3.1.1. Querying the mesh data structure in the finite element analysis procedures is accomplished through the SCOREC/core library which abstracts meshing operations and supports several backends. The numerical integration routines, degree of freedom numbering, and tensor field creation/manipulations are also handled through core/apf.

Interactions with the linear system take place through the LAS interface (discussed in chapter 5), configured to use PETSc [23]–[25] as the backend linear system library, configured to use the `SuperLU_dist` direct LU solver [50]–[52]. A direct linear solver is used rather than an iterative solver since in most cases the size of the finite element model is small enough that a parallel direct solve is effective. As the macro-scale parallel unstructured meshes grow this will become less efficient and a switch to an iterative method will be made.

High-level simulation and analysis structures, along with all multi-scale interactions, are handled by the AMSI/analysis and AMSI/multiscale libraries, respectively. AMSI/analysis provides C++ classes for abstract nonlinear convergence operations, pseudo-timestepping, and extensions/utility operations for core/apf fields operations. Additionally, AMSI/analysis provides functionality to retrieve, create, and apply boundary conditions described through the Simmetrix simmodeler analysis case specification language and tools.

3.2.1.2 Micro Scale Implementation

The micro-scale fiber networks are based solely on discrete meshes, with no explicit geometric domain. These meshes are generated by creating Delaunay (or Voronoi) networks in 3d with customized MATLAB scripts. The fiber properties may be specified on a per-

element basis using an accompanying parameters file defining both the element member type, the fiber constitutive model, and all required physical parameters for the constitutive model. In the multi-scale simulation a library of fiber networks (contained in a single directory) are specified on a per-geometric region basis.

These simple mesh and parameter files are used to create a core/mds edge-and-vertex only mesh, with a first-order Lagrange vector field modeling displacements $\Delta u^{(m)}$ using the core/apf library. An additional mesh representing only the RVE cube is created dynamically to bound the fiber network, which are clipped to the cube dimensions during creation. Separating the unit cube of the RVE from the fiber network mesh simplifies many operations which take place on one object or the other, as few operations are applied uniformly to each mesh.

A core/mds macro-scale mesh of a single element is created using initialization data from the macro-scale domain meshing attributes which describes the element type, shape functions and their order, and the mesh entity parameters for the macro-scale element containing the numerical integration point associated with the local micro-scale simulation. The physical scale of this mesh is used for dimensionalization and calculation of the micro-to-macro coupling terms.

The fiber network mesh is used to create a compressed-sparse row (CSR) sparse matrix data structure for each micro-scale material point using algorithms provided by the LAS (see chapter 5) library. This sparse matrix is assembled into using the elemental system for each fiber generated by a numerical integrator derived from a core/apf integrator, particular to the constitutive type and member type of the fiber edge. After assembly the matrix and load vector are supplied to the LAS interface to solve the linear system, the backend of this instance of the LAS library is configured to use the Sparskit [53] incomplete LU solver. Using a Newton-Raphson nonlinear convergence procedure, the micro-scale will reformulate the micro-scale tangent stiffness matrix and force vector until convergence of the nonlinear iterations.

Sparskit operates only in serial to solve the linear system on the local process. As the fiber network complexity scales it will eventually reach a point where the micro-scale simulations must also be parallelized. Accommodating this from the perspective of the linear algebraic system is trivial as the backend can be immediately swapped out with an alternative parallel linear solver such as PETSc.

3.2.2 Coupling Terms

The macro-to-micro coupling term is the deformation gradient F at the point in the macro-scale domain associated with a micro-scale RVE domain. This is calculated from the set of displacement degrees of freedom effecting each element, at the location of the numerical integration point(s) within the element.

The micro-to-macro coupling terms are a Cauchy-like stress terms and stress derivative terms calculated with respect to the degrees of freedom of the element in the macro-scale domain – each micro-scale simulation has a minimal representation of the single element in the macro-scale mesh to which it is related, created during initialization. The number of stress derivative terms is determined by the macro-scale shape function defining the location and number of degrees of freedom effecting an element, and the type of element at macro-scale. As noted in 3.1.1 all extant meshes contain only tetrahedral elements. The use of meshes with multiple element types would require the creation of multiple couplings, one for each element type, as the distribution of coupling data would differ for each element type.

3.3 Scale Coupling Using AMSI

Listing 3.1: Biotissue Scale-main Functions

```
1  int run_micro_fo(int argc, char * argv[], MPI_Comm cm)
2  {
3    // seed the random number generator since
4    //  micro *can* randomly select RVEs from
5    //  the provided libraries
6    bio::MultiscaleRVEAnalysis rves;
7    rves.init();
8    return rves.run();
9  }
10 int run_macro(int argc, char * argv[], MPI_Comm cm)
11 {
12   // read amsi-specific command-line arguments
13   // initialize configured analysis backends on the provided MPI_Comm
14   amsi::initAnalysis(argc,argv,cm);
15   // load geometry and mesh and initialize analysis case
16   bio::MultiscaleTissueAnalysis an(model,mesh,analysis_case,cm);
```

```
17    an.init();
18    int rslt = an.run();
19    amsi::freeAnalysis();
20    return rslt;
21  }
22  int main(int argc, char * argv[])
23  {
24    // initialize the amsi multiscale components using the configuration
        file
25    amsi::initMultiscale(argc,argv);
26    // set the main-like function for macro-scale
27    amsi::setScaleMain("macro",&run_macro);
28    // set the main-like function for micro-scale
29    amsi::setScaleMain("micro_fo",&run_micro_fo);
30    // execute the multi-scale simulation
31    int rslt = amsi::execute();
32    amsi::freeMultiscale();
33    return rslt;
34  }
```

Using AMSI to support the scale coupling operations required by the soft tissue problem is relatively unobtrusive on the existing single-scale codebases.

As seen in listing 3.1, the main function for the multi-scale biotissue simulation initializes the AMSI multi-scale components using a provided auto-generated configuration file generated by a runtime script. The main function then sets the scale-main functions for the macro-scale and the micro-scale and executes the multi-scale simulation. Both the macro- and micro-scale perform initialization operations and then begin primary execution.

The macro-scale initialization routines initialize the multi-scale coupling by creating a `CouplingData` data structure to describe the parallel distribution of the RVE embedded locations in the macro-scale domain. This `CouplingData` is then used to create and reconcile a `Coupling` with the micro-scale code before finally waiting for the micro-scale to reconcile the return `Coupling`. Refer to listing 3.2 line 6 for this coupling initialization.

The micro-scale initialization routines initialize the multi-scale coupling by creating an empty `Coupling` based on no `CouplingData` in order to reconcile the `Coupling` created by the macro-scale. Once this coupling is reconciled the micro-scale creates an inverted coupling to use the same parallel communication patterns in reverse for the micro-scale to

macro-scale coupling. Refer to listing 3.3 line 6) for this coupling initialization.

After the initialization of each scale is complete, the macro-scale begins normal execution. At the beginning of every pseudo-timestep the `Coupling` is updated based on any migration routines at the micro-scale and decisions made at the macro-scale to create or delete RVEs at specific macro-scale locations based on some criteria. When new RVEs are created (as in the very first time step), the macro-scale creates a delta `Coupling` that describes the changes between the previous `Coupling` and the new `Coupling` after instantiating new RVEs. This delta coupling is used to communicate the initialization data for the newly-created RVEs to the micro-scale, see listing 3.2 line 26 for the macro-scale side of this operation and listing 3.3 line 26 for the micro-scale side of this operation.

After removed RVEs are deleted and new RVEs are instantiates, the boundary conditions are applied at macro-scale, making modifications to the linear system if needed, and an initial guess for the current psuedo-timestep is generated. Just prior to entering the section of code which applies numerical integration to each element in the mesh and assembles the global linear system, a function which calculates and serializes all deformation gradients for the local mesh part is called, followed by a call to the `Coupling` operation, providing the serialized buffer of coupling data (see listing 3.2 line 73). After sending the coupling data, the macro-scale immediately initiates the micro-to-macro coupling to wait for results from the micro-scale.

Listing 3.2: Biotissue Macro-scale Coupling and Driver Functions

```
1  void MultiscaleTissueAnalysis::init()
2  {
3    initCoupling();
4    TissueAnalysis::init();
5  }
6  void MultiscaleTissueAnalysis::initCoupling()
7  {
8    amsi::CouplingData * rve_dist = amsi::createCouplingData(macro,
9                                                             "micro_fo_rves
       ");
10   (*rve_dist) = 0;
11   amsi::assemble(rve_dist);
12   send_ptrn = amsi::createCoupling(rve_dist,macro,micro_fo);
```

```
13    amsi::reconcile(send_ptrn);
14    recv_ptrn = amsi::createRecvCoupling(micro_fo,macro);
15    amsi::reconcile(recv_ptrn);
16  }
17  void MultiscaleTissueAnalysis::updateMicro()
18  {
19    // locally decide whether a micro-scale RVE
20    //  should be created/destroyed/persist/change type
21    //  at each integration point
22    updateRVETypes();
23    // inform micro-scale of deletion/creation of RVEs
24    updateRVEExistence();
25  }
26  void MultiscaleTissueAnalysis::updateRVEExistence()
27  {
28    // collect all rves to be deleted
29    updateRVEDeletion(to_delete);
30    // communicate the deletions to micro
31    amsi::removeData(send_ptrn,to_delete);
32    // serialize all new rve data into buffers
33    serializeNewRVEData(new_headers,new_params,new_init_data);
34    // communicate the additions to micro
35    auto delta_ptrn = amsi::addData(send_ptrn,to_add);
36    // used the delta coupling to send new rve data
37    amsi::communicate(delta_pattern,
38                      new_headers,
39                      mpi_type<micro_fo_header>());
40    amsi::communicate(delta_pattern,
41                      new_params,
42                      mpi_type<micro_fo_params>());
43    amsi::communicate(delta_pattern,
44                      new_init_data,
45                      mpi_type<micro_fo_init_data>());
46    // update the recving pattern for the new RVEs
47    amsi::reconcile(recv_ptrn);
48  }
49  void MultiscaleTissueAnalysis::run()
50  {
51    computeInitGuess();
```

```
52   bool sim_completed = false;
53   while(!sim_completed)
54   {
55     // as part of the iteration process, call the
56     // MultiscaleTissueAnalysis::assemble() function
57     if(amsi::numericalSolve(itr,cvg))
58     {
59       checkpoint();
60       if(step == max_step)
61         sim_completed = true;
62       else
63         step();
64     }
65     else
66     {
67       // failed to converge, handle it
68     }
69   }
70 }
71 void MultiscaleTissueAnalysis::assemble()
72 {
73   computeRVEs();
74   // apply nuemann bcs
75   // elemental integration using rve results
76   // linear system assembly
77 }
78 void MultiscaleTissueAnalysis::computeRVEs()
79 {
80   serializeRVEData(micro_data);
81   amsi::communicate(send_ptrn,
82                     micro_data,
83                     amsi::mpi_tpe<micro_fo_data>());
84   amsi::communicate(recv_ptrn,
85                     micro_results,
86                     amsi::mpi_type<micro_fo_results>());
87 }
```

The main micro-scale simulation loop can be seen in listing 3.3 on lines 52–112, where the outermost loop corresponds to macro-scale simulation psuedo-timestep, and the inner

loop corresponds to the macro-scale simulation Newton-Raphson iteration. Upon receiving a buffer of locally-assigned RVEs in the correct order, each RVE is solved serially (listing 3.3 lines 68-90). First the deformation gradient is applied to deform the corners of the dimensionless RVE cube using the dimensionality of the abstract RVE in the macro-scale domain. After converging to a solution for all local RVEs, the micro-scale produces a set of stress values and derivatives for use by the macro-scale. These terms are packed into a buffer in the same order the macro-to-micro terms were received and sent back to macro-scale (listing 3.3 line 92 for micro-scale and listing 3.2 line 86), while micro-scale waits for the next set of deformation gradients to continue (listing 3.3 line 66).

Listing 3.3: Biotissue Micro-scale Coupling and Driver Functions

```
1  void MultiscaleRVEAnalysis::init()
2  {
3    initCoupling();
4    initAnalysis()
5  }
6  void MultiscaleRVEAnalysis::initCoupling()
7  {
8    // create an empty coupling from macro->micro
9    recv_ptrn = amsi::createRecvCoupling(macro,micro_fo);
10   // reconcile the coupling information from macro
11   amsi::reconcile(recv_ptrn);
12   // create a coupling from micro->macro by inverting
13   //  the coupling which was just received
14   send_ptrn = amsi::invertCoupling(recv_ptrn);
15   // reconcile the coupling information to macro
16   amsi::reconcile(send_ptrn);
17 }
18 void MultiscaleRVEAnalysis::initAnalysis()
19 {
20   // receive information from macro about the number
21   //  of rve/fiber network libraries and their directories
22   //  read in each library and create the fiber networks
23   //  determine the size of the working buffers for the linear
24   //  solver system (Sparskit)
25 }
```

```
26 void MultiscaleRVEAnalysis::updateCoupling()
27 {
28   // receive information on any RVEs which have been deleted
29   amsi::removeData(recv_ptrn,to_delete);
30   // delete the rves
31   ...
32   // receive information on any RVEs which have been added
33   //  create and reconcile a new coupling which only describes the
34   //  data which has been added
35   auto delta_ptrn = amsi::addData(recv_ptrn,to_add);
36   // receive header, parameter, and initialization data for each new RVE
37   amsi::communicate(delta_ptrn,
38                     rve_headers,
39                     amsi::mpi_type<micro_fo_header>());
40   amsi::communicate(delta_ptrn,
41                     rve_parameters,
42                     amsi::mpi_type<micro_fo_params>());
43   amsi::communicate(delta_ptrn,
44                     rve_init_data,
45                     amsi::mpi_type<micro_fo_init_data>());
46   // create new RVE analyses using the received data
47   ...
48   // update the micro->macro pattern to reflect the new RVEs
49   send_ptrn = amsi::invertCoupling(recv_ptrn);
50   amsi::reconcile(send_ptrn);
51 }
52 int MultiscaleRVEAnalysis::run()
53 {
54   int rslt = 0;
55   int sim_complete = 0;
56   while(!sim_complete)
57   {
58     int step_complete = 0;
59     while(!step_complete)
60     {
61       // update the multi-scale coupling, creating/delete
62       //  as dictated by the macro-scale
63       updateCoupling();
64       amsi::communicate(recv_ptrn,
```

```
65                        macro_data,
66                        amsi::mpi_type<micro_fo_data>());
67      int ii = 0;
68      for(auto rve = ans.begin(); rve != ans.end() ++rve)
69      {
70        // use the data from macro-scale to apply boundary
71        //  conditions to the RVE and fiber network
72        applyMultiscaleCoupling(&rve,&macro_data[ii]);
73        // create an object encapsulating the operations
74        //  needed to calculate a single numerical convergence
75        //  iteration on the fiber network simulation
76        FiberRVEIteration itr(*rve);
77        // create an object encapsulating the convergence
78        //   criteria for the newton-raphson procedure:
79        //   fabs(f_{i-1} - f_{i}) < eps (1e-8)
80        RelativeNormConvergence cnvrg(&itr);
81        // perform the newton-raphson procedure on the
82        //  rve until convergence
83        rslt = amsi::numericalSolve(&itr,&cnvrg);
84        // detect any errors (rslt != 0) and handle them
85        ...
86        // use the converged micro-scale state to generate
87        //  results for the macro-scale simulation
88        recoverMultiscaleResults(*rve,&results[ii]);
89        ++ii;
90      }
91      // send the micro-scale results back to macro-scale
92      amsi::communicate(send_ptrn,results,amsi::mpi_type<micro_fo_results
    >());
93      // increment the macro-scale iteration count
94      macro_iteration++;
95      // macro-scale broadcasts whether the current
96      //  psuedo-timestep is complete
97      amsi::scaleBroadcast(M2m_id,&step_complete);
98    }
99    // when the pseudo-timestep completes, reset
100   // the macro iteration counter and increment
101   // the macro timestep counter
102   macro_iteration = 0;
```

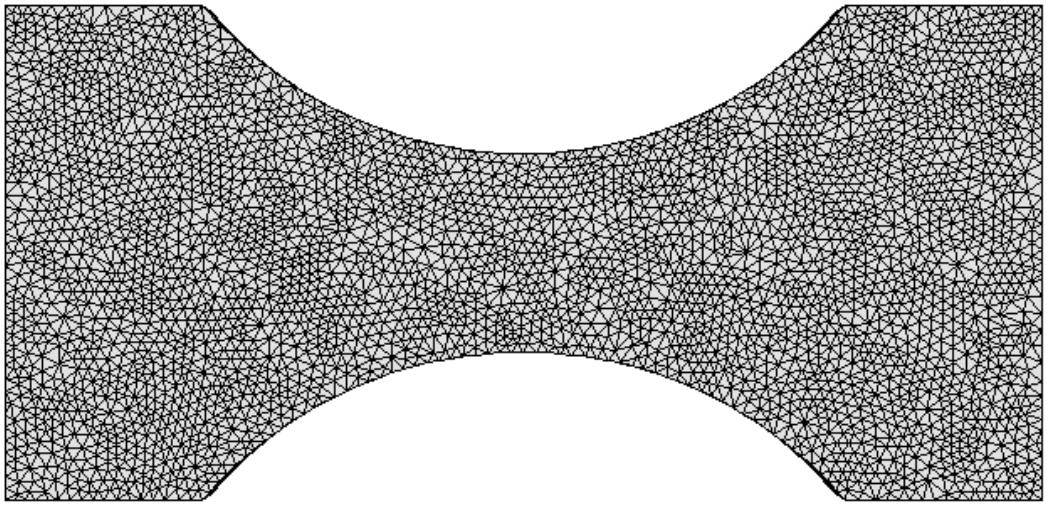

Fig. 3.2: 400k Element Dogbone Tensile Test Mesh

```
103       macro_step++;
104       // macro-scale broadcasts whether the entire
105       // simulation is complete
106       amsi::scaleBroadcast(M2m_id,&sim_complete);
107     }
108     macro_step--;
109     // return whether we were able to handle all
110     //  micro-scale convergence failures
111     return rslt;
112   }
```

The macro-scale receives the coupling data for each numerical integration point in the local domain, and associates the data with the relevant integration point (listing 3.2 line 86). Finally the macro-scale enters the elemental integration procedure, using the supplied stress and stress derivative values in lieu of a constitutive equation to produce an elemental system. From there the macro-scale simulation continues as a standard finite element analysis until the next Newton-Raphson iteration, when the multi-scale coupling process is repeated.

3.4 Example Soft Tissue Problems

3.4.1 Tensile Test Sample

The tensile test sample 'dogbone' domain (see figure 3.2) is a CAD model of a standard materials science and engineering sample used in tensile testing. The sample structure is well-established and provides a useful testbed for development of parameters for use in problems with other, more complex geometries, since the response of many materials with the tensile sample domain and standard tension boundary conditions are widely available and easy to produce.

The dogbone problem domain and associated simulation cases are used here to analyze computational aspects of the multi-scale code, both with respect to numerical performance and simulation resiliency. Many problems of interest require that the body being simulated reach very high-strain states, which can induce numerical instability and cause the Newton-Raphsone procedure to fail to converge. Figure 3.3 shows the macro-scale convergence iteration count (and the linear and quadratic best fits) for each Newton-Raphson process in a simulation that was intentionally run to the point that the Newton-Raphson procedure failed to converge due to an overly deformed mesh. As the simulation proceeded the convergence procedure required additional iterations to converge. As the simulation is designed to regularly checkpoint the valid simulation state after the convergence of each Newton-Raphson procedure the simulation results are retained up to the final, failed step.

Figure 3.4 shows the state of a multi-scale simulation on the dogbone structure after the convergence of the first psuedo-timestep. A mesh of approximately 2k tetrahedral elements is used. The average fiber alignment in each micro-scale RVE is plotted as a normalized vector at the location at which the RVE is embedded in the macro-scale domain. In this simulation the domain is held fixed on all three DOFs for each node on the left-most face of the domain set to zero.

A Neumann boundary condition of an x-axis oriented traction is applied to the right-most face of the domain. The fibers align to the loading axis in the first load step, and continue to align more aggressively throughout the simulation (see figure 3.5 for the state after 10 pseudo-time steps, figure and 3.6 for the state after 20 pseudo-time steps).

In each psuedo-time step the loading on the right-most face is incrementally increased to approach the maximum loading specified by the simulation case attributes. In figure 3.3 the loading increments correspond to the simulation psuedo-timesteps. After 10th loading

Fig. 3.3: Macro-scale Body Extension and Newton-Raphson Iterations per Load Increment

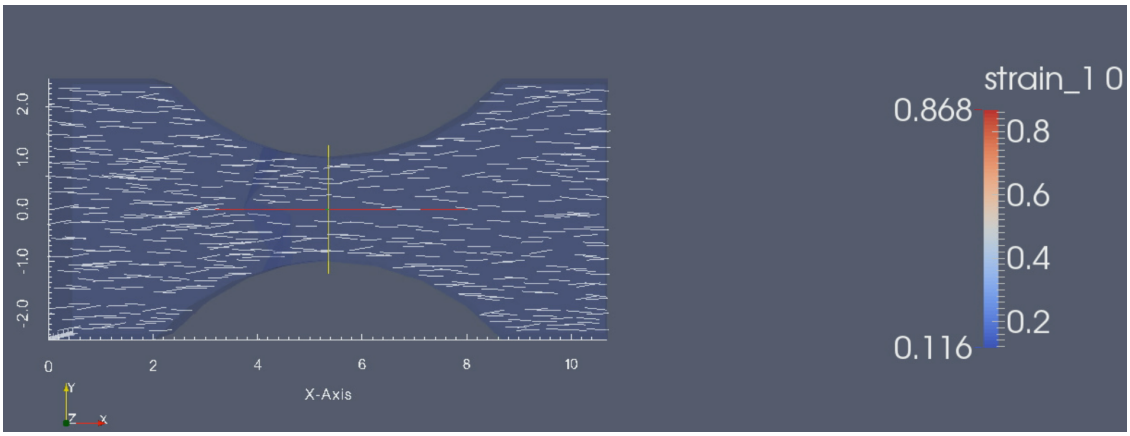

Fig. 3.4: Dogbone 1k Mesh and Fiber Alignment After 1 Load Increment

increment the body has extended by approximately 27% and by the 20th and final loading increment the body has extended by approximately 45% (figures 3.5 and 3.6, respectively).

Figure 3.7 depicts the initial state of an Voronoi fiber network RVE embedded and coupled to the dogbone simulation seen in figure 3.4. The line segment plotted in the center of the RVE cube is the primary fiber alignment of the fiber network, corresponding to the line segments seen in figures 3.4, 3.5, and 3.6.

In figure 3.8 the same RVE is seen fully deformed after 20 macro-scale pseudo-timesteps

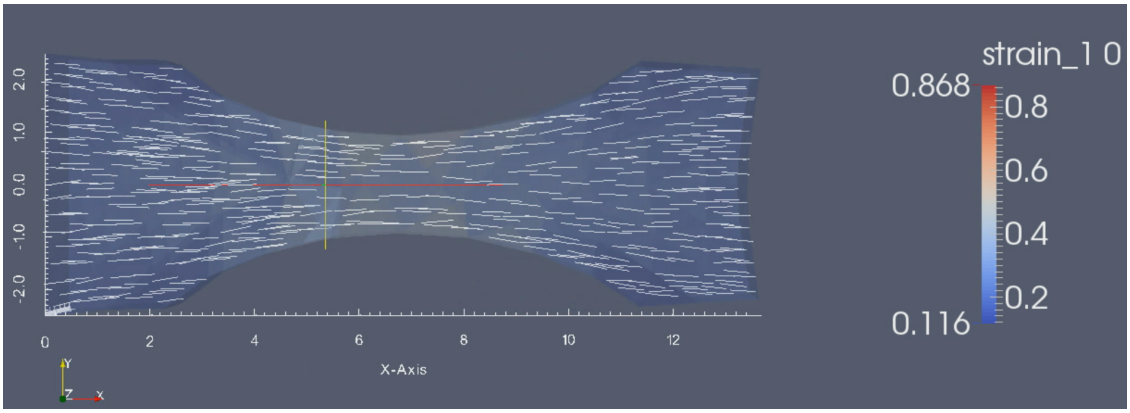

Fig. 3.5: Dogbone 1k Mesh and Fiber Alignment After 10 Load Increments

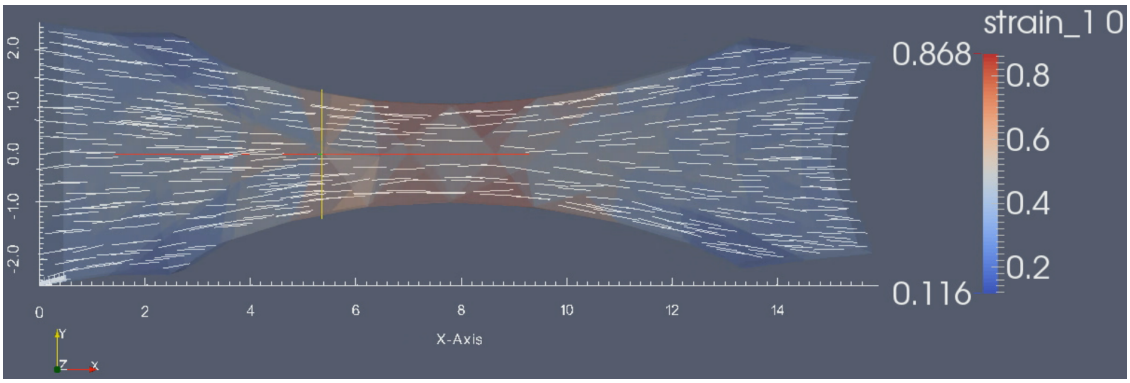

Fig. 3.6: Dogbone 1k Mesh and Fiber Alignment After 20 Load Increments

have converged. The RVE body has extended approximately 60%. In general it is not unusual to see local strains on the RVEs of 2-3 times the total macro-scale body extension. These large local strains can cause stability issues at the micro-scale and require many iterations for the nonlinear procedure to convergence. A more robust convergence procedure (or switching to an explicit solver under certain circumstances) may alleviate this particular bottleneck and allow pushing overall simulations to larger strains.

3.4.2 Glomerular Podocyte

A model of a glomerular podocyte domain can be seen in figure 3.9. This domain was constructed from sample data taken from an actual podocyte cell using the Simmetrix model construction tools [43]. The domain represents half of the cell, the other half would be the mirror of the object across the cut plain most visible on the top surface of the right

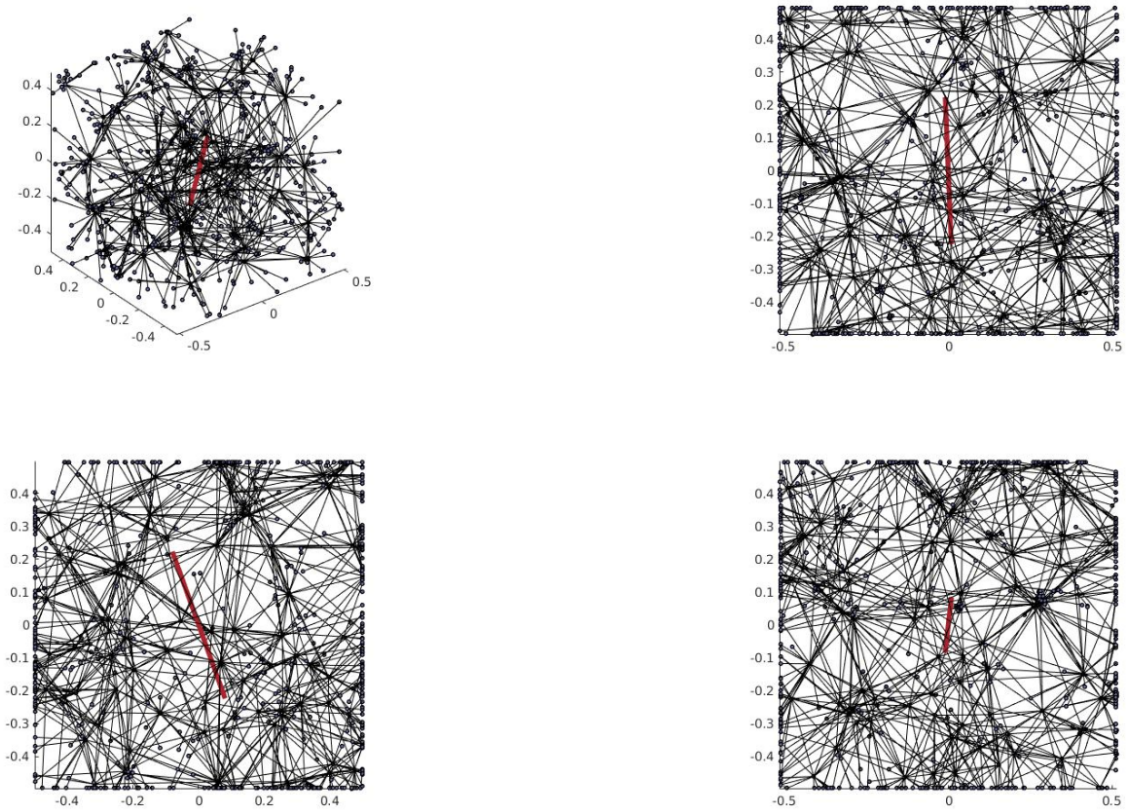

Fig. 3.7: State of an RVE in a Dogbone Anlaysis After a Single Timestep

sub-figure in figure 3.9.

The problem domain has a capillary (the yellow portion of the domain) through which blood is flowing, inducing an internal pressure. The green and blue sections of the domain are the nuclei of the two podocyte cells attached to the artery, while the red regions are the rest of the cell bodies.

In this problem the symmetric face was fixed with respect to the normal direction of the face, while the internal surfaces of the artery were loaded with a normal pressure. A minimum number of degrees of freedom on the symmetric face to eliminate rigid body motions were also fixed.

The capillary portion of the domain was added later in the development process of this particular domain. An initial simulation of the segmented image data without the capillary structure was also conducted (see figure 3.10). The particular problem specification for this domain necessitated the development of pressure boundary conditions which were developed as an extension to the preexisting traction boundary conditions. The model faces on the

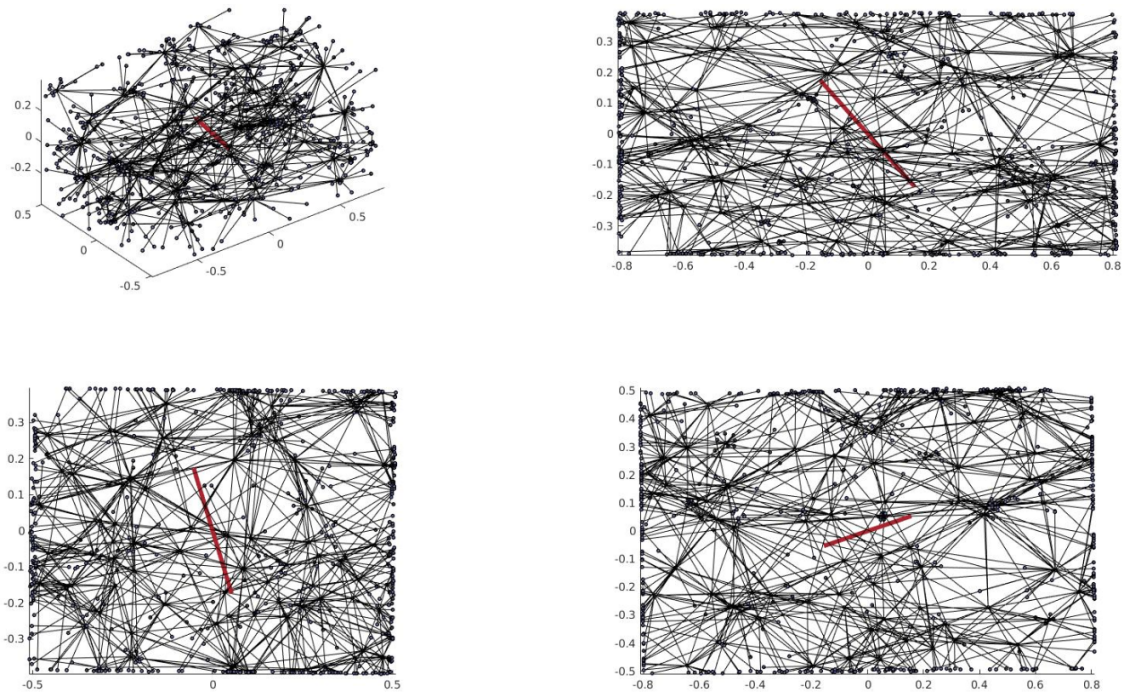

Fig. 3.8: RVE Deformation after 20 Load Increments

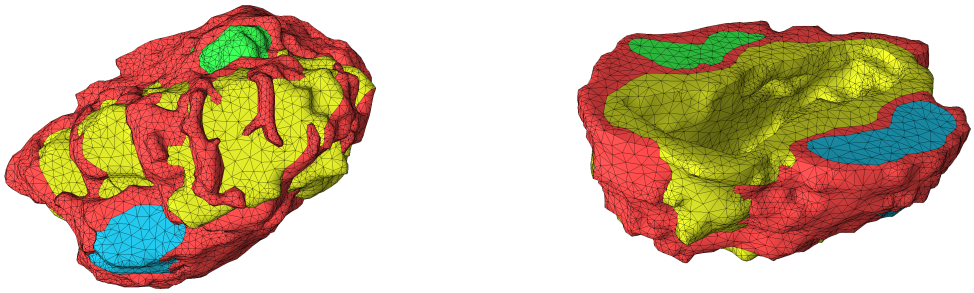

Fig. 3.9: Glomerular Podocyte Mesh

'interior' of the domain – those model faces in contact with the capillary walls in figure 3.9 – were loaded with a pressure boundary condition. This resulted in the cellular 'fingers' of the podocyte cells spreading apart as seen in figure 3.10.

The specification of the podocyte problem case is ongoing as the initial segmented image data used for the construction cellular structures was sufficiently coarse as to introduce artifacts into the resulting model geometry that acted as stress concentrators. These stress concentrators resulted in unexpected and unrealistic behaviors during simulation and it was

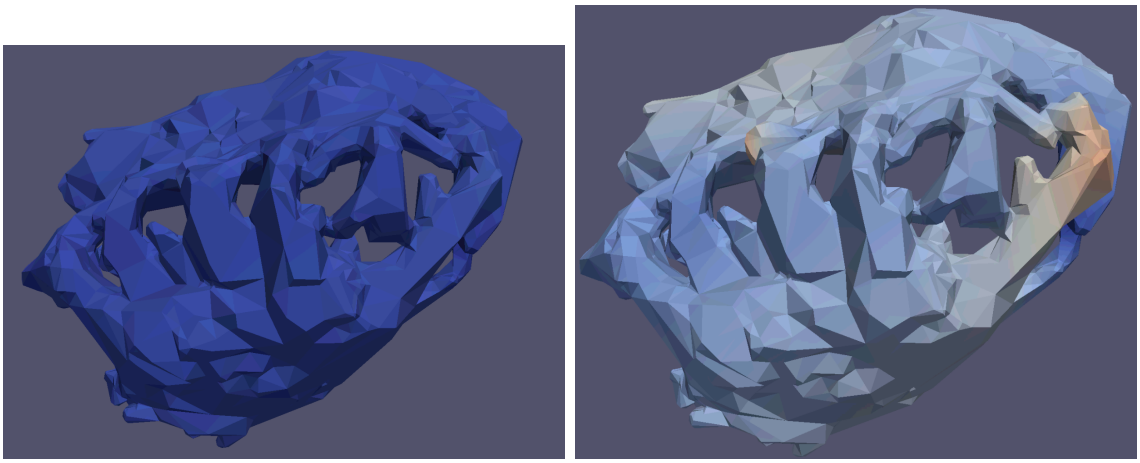

Fig. 3.10: Early Glomerular Podocyte Mesh Pre-loading and Post-loading

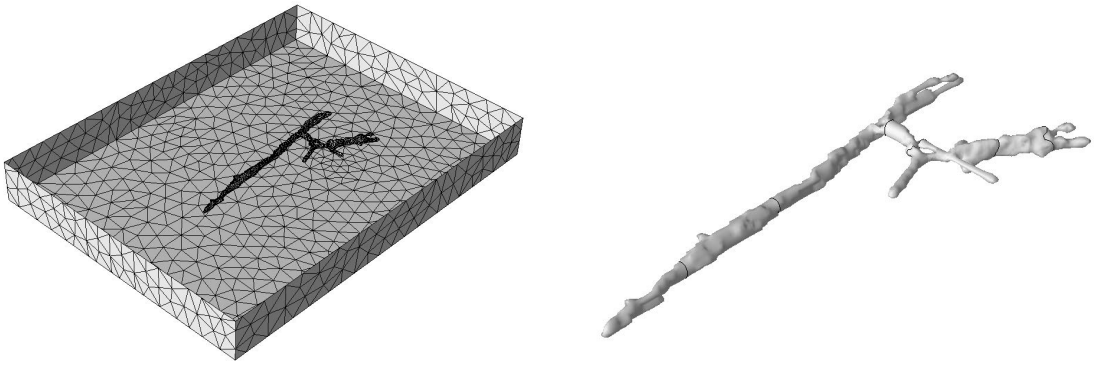

Fig. 3.11: Neuron Mesh

determined a newer domain without these artifacts is needed.

3.4.3 Collagen Embedded Neuron

A neuron domain embedded in a collagen gel can be seen in figure 3.11. The left sub-figure shows the neuron with respect to the collagen gel block it is embedded in, while the right sub-figure shows the isolated neuron. Similarly to the podocyte domain, this structure was created from actual cell data using Simmetrix model construction tools [43]. The domain contains three neuron structures, the most predominant neuron runs axially through the domain, while the other two connect transversely to the predominant neuron.

The experiment most typically conducted on this domain is a uniaxial extension setup, where the collagen gel is held on one end and the opposite face has an applied normal

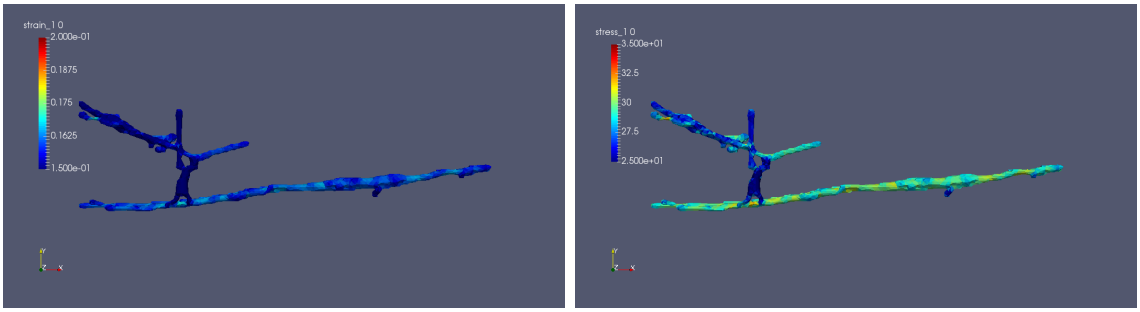

Fig. 3.12: Neuron Pre-loading and Post-loading

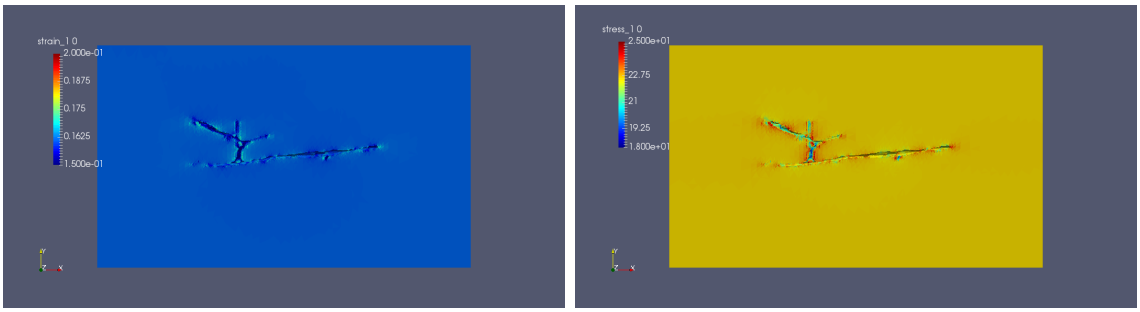

Fig. 3.13: Neuron Gel Pre-loading and Post-loading

traction.

This problem necessitated the development of additional features for the overall simulation. Firstly, the incorporation of multiple fiber libraries exhibiting differing features, as the neuron fiber networks are strongly axially aligned while the collagen fibers networks are typically isotropic. Additionally, the neuron structure should retain global volume throughout the simulation. This necessitates the implementation of a volume constraint.

3.5 Evaluation of Parallel Implementation

All results in this section were generated on the AMOS BlueGene/Q supercomputer managed by the Center for Computational Innovation at RPI. The AMOS BG/Q is a 5-rack, 5120 node supercomputer where each node has a 16 core 1.6Ghz A2 processor and 16GB of DDR3 memory. The IBM xl V12.0 C and C++ compiler suite was used to compile and link the biotissue code.

Strong and weak scaling studies were conducted using the dogbone domain discussed in section 3.4.1.

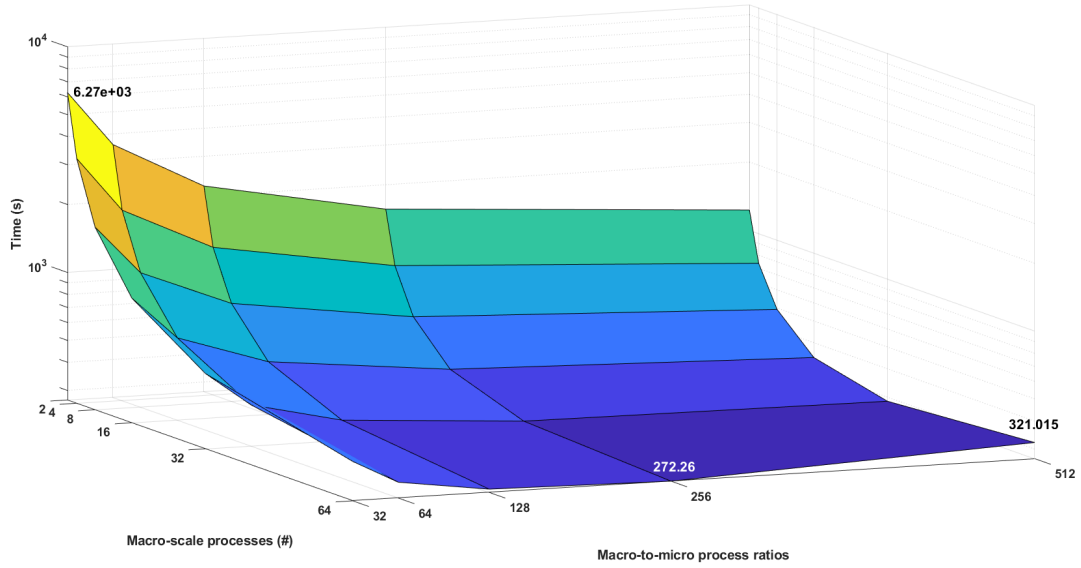

Fig. 3.14: Strong Scaling of the Soft Tissue Multi-scale Simulation

For the strong scaling study a mesh consisting of 64k elements was used, while the number of processes assigned to the simulation was scaled, the results are shown in figure 3.14. In each case the 2nd macro-scale pseudo-timestep of the simulation was evaluated to avoid possible noise introduced by several initialization processes in the simulation that effect only the first pseudo-timestep.

The number of processes assigned to compute the macro-scale simulation varied by powers of two from 2 to 64. The number of processes assigned to the micro-scale was determined as a ratio of the number of macro-scale processes, varying by powers of two from 32 to 512. Thus the processes applied to the entire simulation varied from 64 in the smallest case to 32,768 in the largest case.

Varying the number of macro-scale processes has the greatest impact on the total runtime of the simulation, likely primarily due to the impact the macro-scale increase has on the total number of micro-scale processes, as the distribution and calculation of independent RVEs is embarrassingly parallel. In the worst case, with only two processes assigned to the macro-scale and a ratio of 32, the pseudo-timestep required 6.27e+03 seconds to compute. The best case occurred with 64 macro-scale processes and a ratio of 256, where the compute time was only 272.26 seconds. In the largest case of 64 macro processes and a micro-scale ratio of 512, the compute time was larger than with a 256 ratio. This is likely due to over-

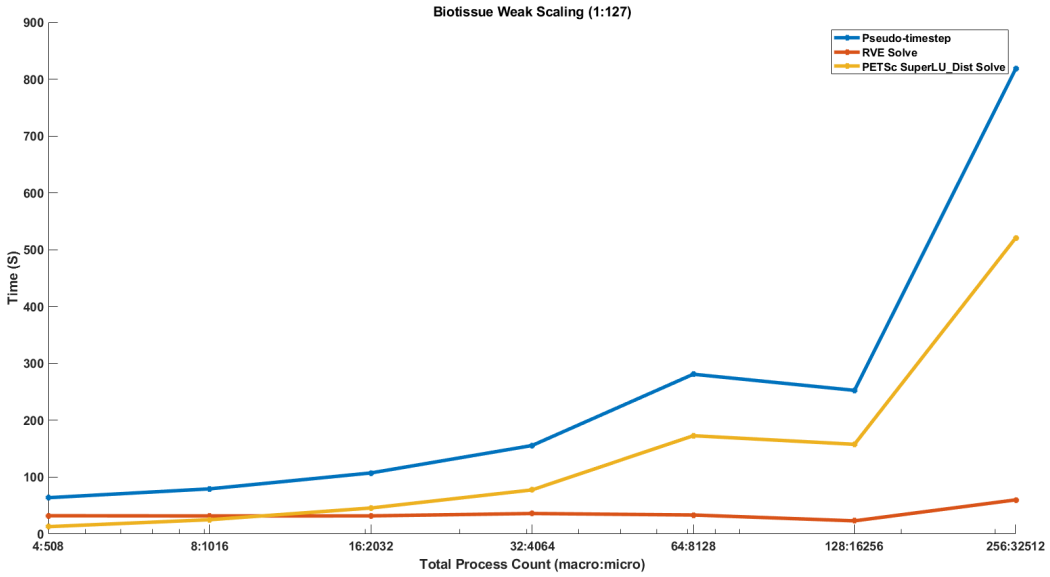

Fig. 3.15: Weak Scaling of the Soft Tissue Multi-scale Simulation

distributing the work as a total of 32,768 processes were assigned to the simulation. As there are 64k total RVEs most micro-scale processes are evaluating only approximately 2 RVEs each macro-scale iteration. Small amounts of noise on the compute nodes, the use of blocking communication operations, and basic communication overhead likely account for this increase.

For the weak scaling study, the dogbone domain is again used. The macro-to-micro ratio is held fixed at 1:127 processes. The total number of macro-scale elements is approximately 1k elements per macro-scale process, which results in 9-10 RVEs being assigned to each micro-scale process. As the scaling increases the RVE compute time remains stable overall. However the macro-scale linear solve time begins to take longer after scaling to 2048 total processes. The linear solver in use at the macro-scale is the `SuperLU_dist` direct LU solver [50]–[52], configured underneath the PETSc libraries and APIs. This direct solver is ill-suited to solve sufficiently large parallel linear systems and an iterative solver should be used.

Looking at the timings of the same set of operations in the strong scaling case we see that beyond a modes number of processes the parallel direct solve is the predominant limiting factor in the time-to-solution of the problem, quickly plateauing after only 16 macro-scale

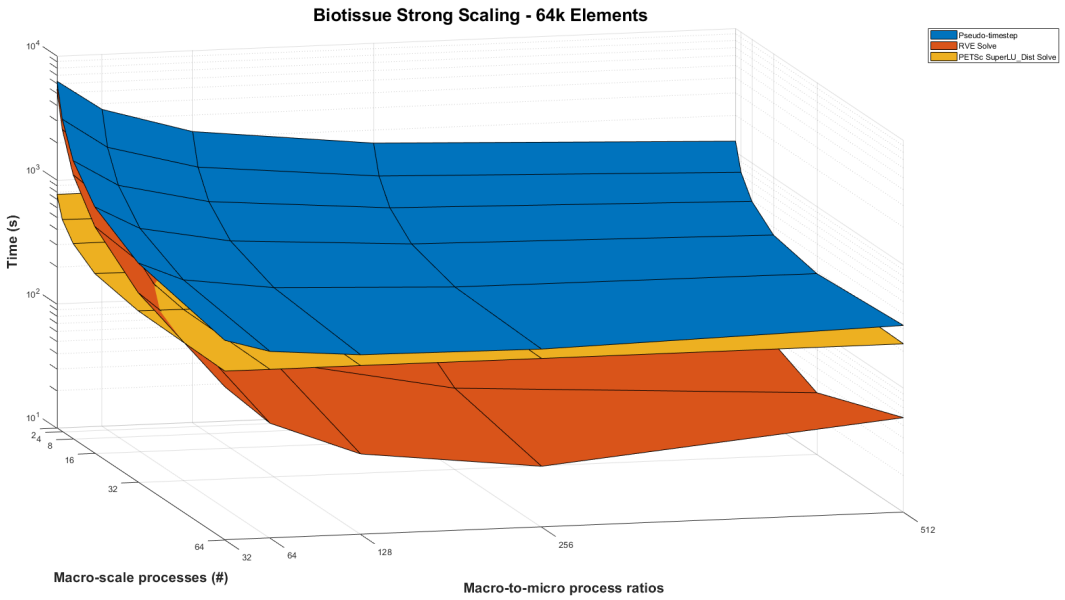

Fig. 3.16: Strong Scaling of the Soft Tissue Multi-scale Simulation – Per Operation

processes are applied to the 64k element problem. As noted in the initial strong scaling analysis above the uptick at the highest level of scaling seen is indeed due to the RVE computation and communication, as the RVEs are spread so thinly throughout the parallel execution space.

3.6 Conclusion

In this chapter we have applied the AMSI multi-scale simulation capabilities introduced in chapter 2 to implement a multi-scale simulation of soft biological tissues called 'biotissue'. The physics underlying the implementation of the two coupled scales in the simulation was briefly described, and is expanded on in appendices A and B. The implementation details of the individual scale codes and their solution processes were discussed. The data structures of the coupling terms used to couple the scales together was discussed. The implementation of the AMSI multi-scale coupling API in each of the simulations was discussed. The application of the biotissue simulation software to several problems of interest including a tensile test sample, glomular podocyte cell structure, and neuron embedded in a collagen gel was discussed. The parallel characteristics of the biotissue simulation was evaluated and shown to display good strong and weak scaling characteristics, and limits to the current parallel

implementation were considered.

CHAPTER 4
SCALE-SENSITIVE LOAD BALANCING OPERATIONS IN HIERARCHICAL MULTI-SCALE SIMULATIONS USING AMSI

4.1 Load Balancing

As problem sizes grow - whether in terms of the underlying physical model used to simulate the problem or the granularity of the domain discretization or problem domain itself - single-scale simulations must be conducted on massively parallel machines in order to achieve meaningfully responsive results. This is accomplished by distributing problem data across the execution space of the machine, and keeping communication/blocking points to a minimum.

In order to maximize the computational gain granted by parallelization, load balancing must be used. The general load balancing problem depends on the implementation of the model used in a simulation; in terms of the data/tasks to be balanced as well as the necessary coordination between processes in order to advance the simulation. Further, the characteristics of the machine must be taken into account, both with respect to the hetero/homogeneity of the computational resources of the machine and the communications facilities employed. Finally combining the other dependencies, the load being balanced must have some metric denoting the computational demand associated with the discrete units capable of being distributed across the machine, as well as (possibly) the communication cost associated with that distribution. For a more in-depth discussion of load balancing theory, see [54]–[57].

Dynamic load balancing is a difficult problem, though many widely-used numerical methods and/or specific implementations of methods have developed relatively mature load balancing algorithms or heuristics [56]. In particular the finite element method – used in the primary scale of the Biotissue problem – has a mature set of load-balancing methods based on graph methods [58]–[61] used to distribute entities of the discretized domain and associated tensor field values across the execution space.

Briefly, the computational demand on each process can be estimated as a function of the number of individual elements that must be processed by each node, but due to dynamic

mesh adaptation processes used to minimize error estimators across the numerical problem domain, the global number of elements associated with the discretization may vary wildly over the course of a multi-load step simulation [62]. Dynamic load balancing operations allow for the redistribution of individual elements between processes after adaptive processes have created an imbalance in the element distribution, achieving a more balanced element count and resulting in better overall process utilization.

4.2 Multi-scale Load Balancing

Using numerical methods for which load balancing schemes have been developed in a multi-scale simulation introduces a new set of constraints to the problem due to the inter-scale dependencies. Adaptive operations which completely invalidate the geometric relationship and consequently the multi-scale coupling terms between two scales in a multi-scale simulation requires reinitialization of the geometric relationship and coupling.

When adaptive operations do not invalidate the multi-scale coupling relationship by retaining the domain characteristics that establish the relationship it is possible to take advantage of load balancing operations so long as any load balancing effecting the coupling relationship is modified to accommodate the new parallel distribution of coupling terms. This process is referred to as scale-sensitive load balancing, where a single scale undergoes a load balancing process effecting the coupling characteristics of that scale, and all coupled scales are informed of the redistribution.

4.3 Scale-Sensitive Load Balancing Implementation

The load balancing planning algorithms of Zoltan [63], [64] or other load balancing procedures can be used underlying the scale-sensitive load balancing planning of AMSI. AMSI further provides users with the ability to create and register custom load-balancing algorithms specific to the requirements of a given numerical implementation.

Scale-sensitive load balancing is a three-phase process: a planning phase, migration phase, and scale-syncing phase are required to complete a single load-balancing event.

4.3.1 Phase 1: Planning

The planning phase is a collective operation on the set of processes associated with the scale being load-balanced. During this phase weighting constraints and other information

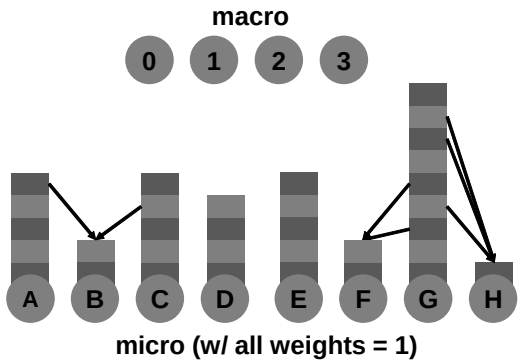

Fig. 4.1: Scale-sensitive Load-balancing Phase 1: Planning

Table 4.1: Simple Load Balancing Plan

```
A:{(4):B}
B:{}
C:{(3):B}
D:{}
E:{}
F:{}
G:{(2,4):F,(3,6,7):H}
H:{}
```

is provided to a planning algorithm which produces a plan for redistributing information such that the weighting metric is balanced. For this phase AMSI typically utilizes planning algorithms provided by the Zoltan library, though as noted above any planning algorithm may be registered and used with the system.

Figure 4.1 depicts a load-balancing plan in a multi-scale system with 2 associated scales. The macro-scale is executing on processes 0-3 and the micro-scale is executing on processes A-H. Each micro-scale process has a number of load-balancing units (depicted as a stack of boxes above the process identifier), each of which is associated with a weight. In this example a uniform weighting of 1 is assumed for all load balancing units (represented by the bars above each process) for simplicity and clarity of discussion. The load balancing plan itself is represented by a set of arrows in the figure. The actual plan corresponding to the depiction is seen in table 4.1 where each process has set of pairs of lists of local unit ids and the foreign micro-scale process to which they are being sent in the load balancing process.

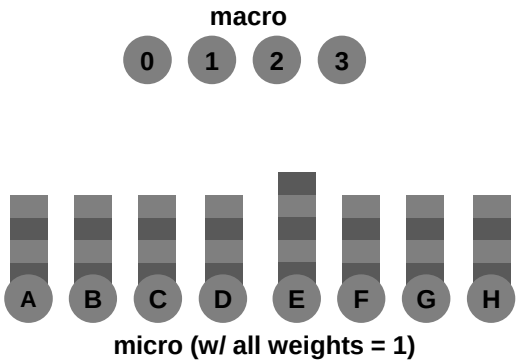

Fig. 4.2: Scale-sensitive Load-balancing Phase 2: Migration

4.3.2 Phase 2: Migration

The migration phase is also collective on the set of processes associated with the scale undergoing load-balancing. During this phase the plan from the previous step is put into action. Each process with a non-null set of migration operations sends the list of local load-balancing units to the receiving process. Serialization of the data on the origin process and deserialization/reinitiatization of the data on the receiving process is often required during this phase as well.

Figure 4.2 depicts the result of implementing the load balancing plan introduced in figure 4.1 during the migration phase. By transmitting units of load balanced data between micro-scale processes, each process has as equal a weight as is possible to achieve given the weight granularity. In this case only a single process, has more weight assigned than any other process. In practice achieving equal weighting through load balancing is often impossible, so the planning routines typically operate by bounding the imbalance present after the load balancing migration takes place.

When using the routines provided by AMSI to enact the migration operations required by a load balancing plan, each unit of coupling data is appended with two pieces of metadata used by the final phase of the scale-sensitive load balancing process. This metadata is the scale-rank of the process from which the load balancing unit data is sent, and the local id of the load balancing unit on that rank in the context of any CouplingData distributions used to derive Couplings on the scale.

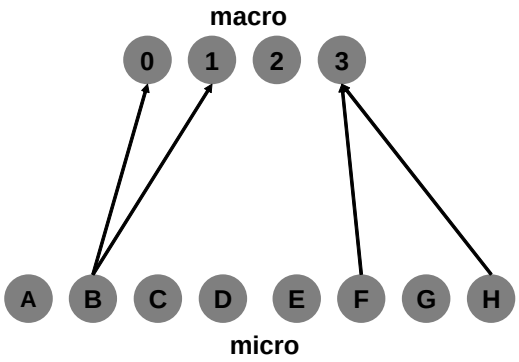

Fig. 4.3: Scale-sensitive Load-balancing Phase 3: Scale-syncing

4.3.3 Phase 3: Synchronization

A scale synchronization operation is collective over the processes assigned to the simulation scale undergoing a load balancing operation and the processes assigned to a coupled simulation scale. The synchronization phase requires a scale synchronization operation for each scale Coupled to the load balancing scale using CouplingData involved in the load balancing operation.

In a synchronization operation the appended metadata redistributed with the migration data is used to provide information to associate scales about the movement of the data at the scale being load-balanced.

While the initial load balancing plan contains sufficient information to inform the local scale about required migration operations, it doesn't provide sufficient information to coupled scales to update coupling patterns. Both the rank-local id on the scale rank sending the load balancing data and the rank-local id on the scale rank receiving the data must be provided to any coupled scales in order to appropriately update any Couplings related to the load balanced units of data. As such the AMSI migration phase appends metadata to load-balanced data (as noted in section 4.3.2) instead of the load balancing plan itself being sent to the coupled scales, as only the scale ranks receiving the load balancing data have sufficient information to fully inform the necessary Coupling updates.

Figure 4.3 shows the point-to-point communications required in the example case being discussed, though as stated above the operation is collective over the union of the two scales as every process may have required communication operations. Each rank which received any units of load balancing data in the migration phase (in this case ranks B, F and H

Table 4.2: A Weighted Coupling Between two Scales

Macro Ranks	Macro cd	Coupling							
0	125	64	61						
1	122		3	64	55				
2	128				8	63	57		
3	132						6	63	63
	Micro cd	64	64	64	63	63	63	63	63
	Micro cd wgt	64	64	64	63	63	64	227	221
	Micro Ranks	A	B	C	D	E	F	G	H

in the micro-scale) now use use the Coupling data structure to determine which ranks in the coupled scale are associated with the pieces of redistributed coupling/load balancing data. These ranks in the coupled scale are sent a message for each migrated unit of coupling data containing the origin rank in the micro-scale, the local id on the origin rank, the receiving rank in the micro-scale, and the new local id of the data on the receiving rank. Using this information the macro-scale Coupling is updated to reflect the new CouplingData distribution at the micro-scale. The Coupling update process is discussed in more detail along with an accompanying example in section 4.4.

4.4 Synchronization: Multi-scale Coupling

During the synchronization phase of a scale-sensitive load balancing operation the scale undergoing the load balancing operation provides information to any foreign scales coupled through the CouplingData being redistributed by the migration phase of the operation. This information is used to modify any Coupling data structures operating on the modified CouplingData. Table 4.2 provides the state of a Coupling data structure associated with a CouplingData with weighted data units undergoing a scale-sensitive load balancing operation.

4.4.1 Initial Coupling Structure

The 'macro' scale plays the sender role and consists of processes 0-3. The 'micro' scale plays the receiver role and consists of processes A-H. A CouplingData (column 'macro cd') on the macro-scale is distributed by the Coupling data structure to the micro-scale, creating an induced CouplingData at the micro-scale (row 'micro cd'). The unit count of CouplingData being sent by a specific macro-scale process `macro_proc` to a specific micro-

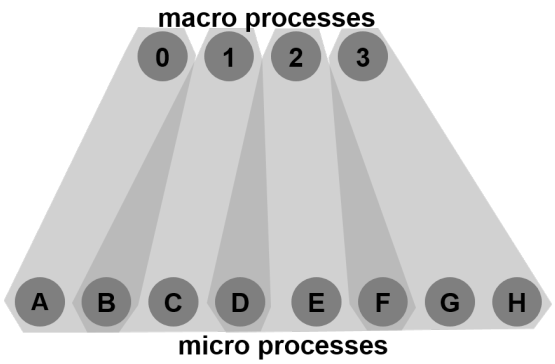

Fig. 4.4: Communication Neighborhoods Arising from a Coupling Pattern

scale process `micro_proc` is given by the value in the CouplingData matrix at location `(macro_proc,micro_proc)`. The local ids of these units of coupling data are determined solely by their location in the communication data buffers used to enact a Coupling operations.

Figure 4.4 depicts the communication neighborhoods induced by the Coupling data structure depicted in table 4.2. Each shaded region encompassing a set of processes in different scales indicates that the processes in each scale engage in point-to-point communication during a Coupling operation. In particular, since rank 0 in the macro-scale is contained in a communication neighborhood with rank A and B in the micro-scale, it will only need to communicate with these ranks during a Coupling operation. This can be seen in table 4.2 as these are the only nonzero locations in the Coupling table representation in the 0th-process row.

In this particular coupling relationship use of the default AMSI coupling algorithm to evenly distributed the coupling data is sufficient, resulting in each micro-scale process receiving either 63 or 64 units of coupling data. This coupling data has no weight assigned to it a-priori, but after a single round of computation at the micro-scale a set of weights are generated (see the emphasized 'micro cd wgt' row in table 4.2), and the weight on processes `G` and `H` are sufficiently disproportional to the weights on the other micro-scale processes that a load-balancing operation is prudent. The imbalance threshold to trigger a load balancing operation is user-determined, often 5-10% is a good heuristic range.

A load balancing plan is generated (see table 4.3) that will result in the weighted coupling data being more well-balanced among the micro-scale. This particular plan is

Table 4.3: Greedy Load Balancing Plan for Weighted Coupling

```
A:{}
B:{(0-29):A}
C:{(0-63):B}
D:{(0-15):B,(16-62):C}
E:{(0-50):C,(51-63):D}
F:{(0-62):D}
G:{(0-4):D,(5-35):E,(36-61):F}
H:{(0-22):G}
```

Table 4.4: A Weighted Coupling After Load Balancing

Macro Ranks	Macro cd	Coupling							
0	125	104	23						
1	122		67	30					
2	128			74	64				
3	132				16	30	26	25	40
	Micro cd	64	64	64	63	63	63	63	63
	Micro cd wgt	104	104	103	106	105	108	100	100
	Micro Ranks	A	B	C	D	E	F	G	H

generated by a greedy load balancing algorithm in order to produce an easily-visualized plan with respect to the Coupling data structure, in practice a plan which minimized data movement as well would be produced by algorithms in Zoltan or another load balancing library.

4.4.2 Modified Coupling Structure

The load balancing plan enumerated in table 4.3 is used during the migration phase on the micro-scale such that the weighting differentials between any two processes in the micro-scale is minimized (see the 'micro cd wgt' row in table 4.4). After the plan is enacted and migration has taken place through the AMSI load balancing interface, the traces of AMSI metadata passed through the load-balancing migration process are used by the AMSI systems to update any coupling patterns related to the specific coupling data being load-balanced. This gives rise to a new coupling pattern which replaces the old coupling pattern, seen in 4.4.

The internal permutation buffer described in algorithm 4.1 is used in subsequent Coupling operations (and updated in subsequent Coupling modifications) so the user in not required to reorder their data in cases where that would be burdensome. However the user

Algorithm 4.1 Coupling Update During Synchronization

```
1: for all update in couplingUpdates do
2:     for all id_f in update.foreignIDs do
3:         id_l ← COUPLING.TRANSLATETOLOCAL(id_f)
4:         COUPLING.REMOVE(update.foreignProcessOrigin, id_l)
5:         id_new ← COUPLING.ADD(update.foreignProcessTarget)
6:         COUPLING.PERMUTE(id_l, id_new)
7:     end for
8: end for
```

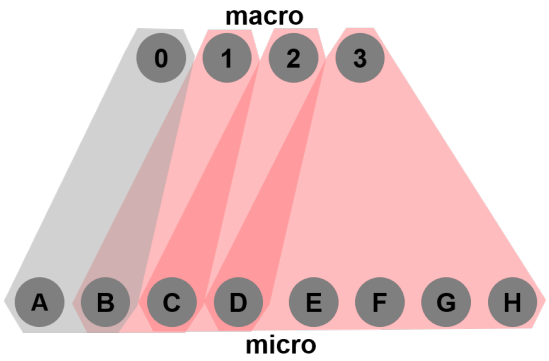

Fig. 4.5: Modified Communication Neighborhoods after load Balancing

can retrieve the permutation array in order to reorder their data structures or to change the serialization ordering of data for use in Coupling communication operations. After retrieval of a permutation buffer AMSI treats all future communication operations involving the related Coupling as though they are unperturbed.

As it is necessary to retain all multi-scale couplings effected by a scale-sensitive load balancing operation, the actual data migration required to implement a specific load balancing plan may take place internally in AMSI. AMSI is provided with serialized buffers of load balancing data associated with an already-planned load balancing operation on each process in the AMSI scale undergoing load balancing.

This new coupling pattern induces new coupling neighborhoods for communicating the coupling information from macro-scale to micro-scale, seen in figure 4.5.

In order to maintain usability in more general situations, it is also possible to simply inform the AMSI system that the current load balancing plan has already been accomplished by user-implemented data migration. This functionality combined with the ability to use user-designed load-balance planning algorithms allows AMSI to work with entirely external

load-balancing libraries and algorithms, allowing scale-sensitive load-balancing to be used even when a specific multi-scale use case falls outside the bounds of capabilities built into the system.

4.5 Application to Hierarchical Multi-scale Soft Tissue Simulation

See chapter 3 for discussion of the structure and implementation of the soft tissue 'biotissue' simulation and code used in our evaluation of scale-sensitive load balancing operations.

Scale-sensitive load balancing operations are conducted on the micro-scale RVE simulation distribution at locations in the code where global inter-scale rendezvous operations already occur. The primary location which fit this requirement are at the end of a macro-scale iteration (just prior to the macro-to-micro coupling, during the coupling update discussed in section 3.3, specifically listing 3.3) or at the end of the macro-scale pseudo-timestep (see section 3.3, listing 3.3).

Conducting load balancing in these locations prevents the introduction of new global rendezvous operations which may reduce simulation performance. Accumulating sufficient historical load-balancing weighting information about the RVE simulations motivated the decision to conduct load-balancing operations at each macro-scale psuedo-timestep rathet than at each Newton-Raphson iteration.

This also has the effect of limiting the overall number of load balancing operations in a simulation, as each load balancing operation itself introduces overhead (see discussion of figure 4.9).

The primary restriction to the redistribution of the RVEs on the micro-scale processes for load balancing purposes is that the relation to a specific macro-scale numerical integration point must be maintained.

As each RVE is associated with a nonlinear simulation there is no a-priori estimate of the computational load each RVE represents, so the optimal initial distribution is simply treating the weight of each RVE as 1. Thus each micro-scale process is assigned an equal number of RVEs ± 1. Once the first load-balancing event is reached, regardless of the granularity at which load-balancing is occurring for a given case, each RVE has recorded the total number of newton iterations it has conducted since the previous load-balancing event. This number is used as a heuristic weighting metric to estimate the computational load of each

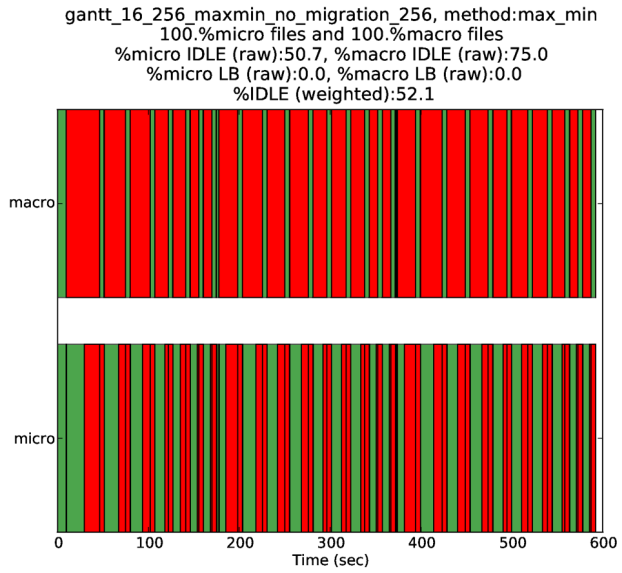

Fig. 4.6: Process Status in Unbalanced Soft Tissue Simulation Over Three Load Steps – Red Sections Represent Time when a Single Process is Idle, Green Sections Represent Time when all Processes are Active

individual RVE for the next micro-scale RVE solution.

The iteration count metric was chosen as each micro-scale nonlinear iteration takes a similar level of work to complete, up to the difference in the number of degrees of freedom for each fiber network, which is minimal. We hypothesized that the newton iterations required for converge at a micro-scale simulation would persist relatively well over the course of an entire multi-scale simulation. Assuming linear incremental loading of the macro-scale domain, regions in the discrete mesh located near stress concentrators would (over the course of the entire simulation) provide a coupling state to related micro-scale RVEs that resulted in a set of micro-scale boundary conditions and an initial guess require more iterations to converge to a solution than those RVEs related to macro-scale elements undergoing little deformation. These RVEs located in high-stress regions would retain a high iteration count over the course of the simulation, while RVEs in low-stress regions would converge relatively quickly, providing us with a persistent weighting metric.

All discussed results are based on simulations of the problem domain seen in figure 3.1, a model of a standard dogbone tensile-test sample.

Figure 4.6 shows the equally-weighted results for three pseudo-time load steps. The macro-scale efficiency and micro-scale efficiency charts run side-by side from wall-clock time

0 to approximately 600. Areas of the chart in red indicate that any process in the scale was idle, while areas in green denote that all processes in the scale were active. Load balancing aims to improve the portion of time the micro-scale is in the idle state by decreasing the process with the longest time-to-solution during each micro-scale convergence process. In this equally-weighted case the micro idle time is 50.7%, while the macro idle time is 75%.

The characteristics of figure 4.6 reflect the simulation phases. While the macro-scale is performing a linear solve, the micro-scale lies idle, and while the micro-scale simulations are performing their own Newton-Raphson iterations each incremental load step, the macro-scale sits idle. The load balancing scheme we use is intended to reduce the time to completion of the final micro-scale simulation to complete, as this is what determines when the macro-scale simulation may begin it's global solve procedure.

Figure 4.7 shows the weighting metric deviation from the norm for 256 processes in the micro-scale over three load steps. Each sub-figure shows the sum of the weights of the RVEs associated with 256 micro-scale assigned processes during the first three psuedo-time steps of the simulation. The intended effect of the load-balancing scheme is to reduce the processes with the largest deviations from the norm, as these represent addition RVE nonlinear iterations, and thus compute time.

Figure 4.9 shows the effect of using scale-sensitive load balancing on the same problem. The first load step is identical as we again use an equally-weighted scheme prior to the first macro-scale pseudo-time step. The load balancing phases on micro-scale can be seen in the figure as blue sections of time. Unfortunately the maximum deviation from the norm in step 2 is seen to increase, though decrease in step 3, as can be seen in figure 4.8. The micro idle percentage also increased to 61.5% while the macro idle percentage decreased slightly to 73.8% (likely as a function of the increased time rather than our load balancing efforts).

These results taken in concert with successive results lead us to believe that an effective load balancing approach for this problem would require a hierarchical load balancing scheme making use of problem over-decomposition. This would assigned work pools of RVEs to different micro-process sets, determined by use of hardware locality tools to be on the same computational node or plane. Thus local balancing would occur as processes in the pool share the computational load, and the above load balancing scheme can be periodically used to balance the load between the various work pools. This represents future work and is further discussed in section 6.2.3. This also has the effect of reducing frequency of global

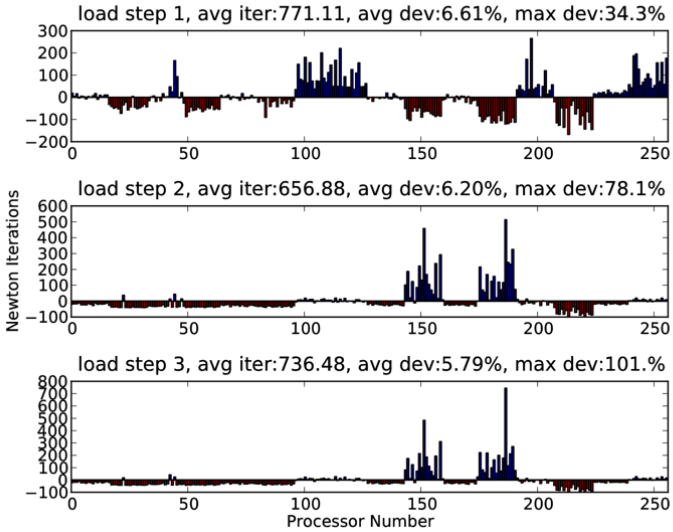

Fig. 4.7: Deviation of Weights on Unbalanced Micro-scale Processes

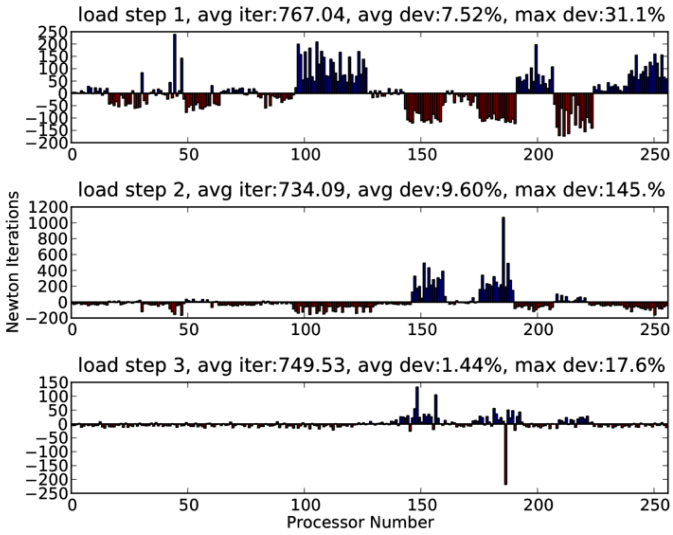

Fig. 4.8: Deviation of Weights on Balanced Micro-scale Processes

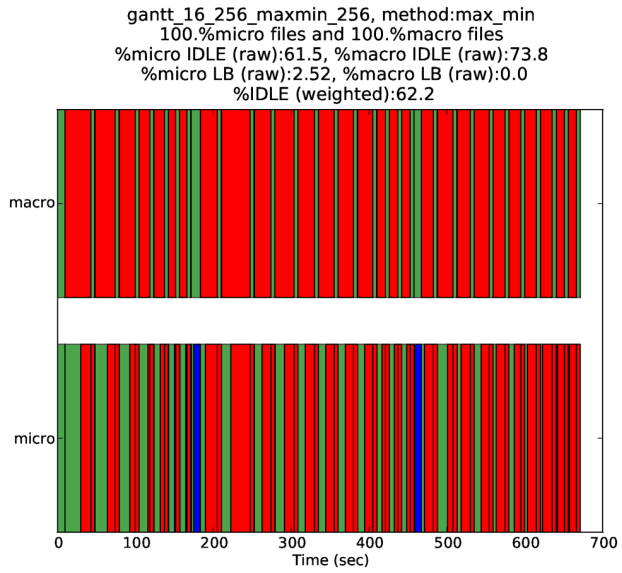

Fig. 4.9: Process Status in Dynamically Balanced Soft Tissue Simulation Over Three Load Steps – Red Sections Represent Time when a Single Process is Idle, Green Sections Represent Time when all Processes are Active, Blue Sections are Time Spent Load Balancing

parallel communication operations used for load balancing, which themselves add to the total solve time of the simulation. The efficiency of the communication processes required for the load-balancing operations is another component that must be considered in additional work on this subject.

4.6 Conclusion

In this work we have developed an approach to support load balancing operations in the context of multi-scale simulations coupled using the Adaptive Multi-scale Simulation Infrastructure (AMSI). We developed this approach as an extension of the standard 2-phase load balancing model of planning-migration through the addition of a third multi-scale synchronization phase. We developed a multi-scale coupling modification capability to enable the synchronization phase. To realize these capabilities, we extended the AMSI implementation to provide both full 3-phase scale-sensitive load balancing support and solely 3rd phase synchronization capabilities to allow usage in a broad set of use cases.

We leveraged our full 3-phase scale-sensitive load balancing operations in the context of a hierarchical multi-scale simulation of soft biological tissues. Further we discussed the char-

acteristics of the load-balanced weighting metric in the soft-tissue simulation and found that, while the load balancing capabilities were effective in producing a more balanced weighting on each process in the micro-scale of the simulation, the underlying weighting metric in use was unable to properly capture the evolving computational expense of the data over the course of a simulation.

CHAPTER 5
IMPLEMENTATION OF ZERO-OVERHEAD ABSTRACTION LAYERS FOR USE IN HPC HIGH-PERFORMANCE LOOPS

5.1 Component Software in HPC

The development of scientific codes for high performance computing (HPC) architechtures is a laborious process requiring the involvement of developers from many differing specializations. The construction of framework software packages for HPC development allows for many benefits including allowing the modular construction of new functionality through the use of components from many developers [65].

Components are intended to provide a set of services to the user to accomplish a task, and many components from different developers exist that provide similar sets of services and can accomplish the same task in a variety of ways. The development of a common interfacing standard to ease the difficulty of combining of software components has long been a goal in the HPC community, seeing the rise of standards such as CCA [66], [67] and GridRPC/Omni RPC [68], [69].

The specific context of multi-scale computing on HPC machines has motivated the development of new component models [70], [71] targeted at creating environments for the construction of multi-scale simulations, as discussed in chapter 2. The Abstract Scale Model and Abstract Coupling model introduced in section 2.4 are abstractions based on a generic component model for individual scale codes.

There are often difficulties in adopting component frameworks including the cost of refactoring existing code, overhead introduced by the framework, and incompatibility of approaches between a framework model and a particular use case. Due to these difficulties it is still common to encapsulate existing libraries in a component wrapper which exposes an interface geared toward a specific set of operations required by the developer. These abstraction layers allow many of the benefits of a full component-based approach without the requirement of refactoring existing codes to accommodate a framework.

5.2 Abstraction Layers

The use of a well-designed abstraction layer can allow a developer a great deal of leeway when designing and implementing algorithms, allowing the decision about the use of a specific underlying package or library to be deferred until compile- and even run-time. Abstraction layers and application programming interfaces (APIs) are a fundamental tool in software engineering, allowing the implementation details for operations to be both obfuscated and altered orthogonally from the implementation of algorithms making use of the abstraction layer.

Further there is support for the assertion that using a proper abstraction layer may result in better code efficiency than specialized, direct usage of a 3rd party component library, in part due to the reduction in the knowledge gap on the part of the component user [72].

Many classes of applications exist where runtime overhead associated with use of API layers is undesirable as it can impact performance and efficiency of the program at a very low-level [73]. These include applications targeting highly-constrained deployment platforms such as embedded systems where compute power and memory size are the primary constraints and high performance computing applications where problem size is the primary constraint. In these constrained situations, evaluating the efficacy of abstraction layer implementation and usage is important [74].

5.3 Runtime vs. Compile Time APIs

The most important feature of an abstraction layer is simply to provide an abstraction of the operation being performed through a generic interface. The determination of the actual concrete function calls corresponding to the usage of the abstraction layer interface is primarily made either at run-time or compile-time [75]. In C++ run-time abstraction is typically accomplished through inheritance polymorphism, though C-style 'abstraction' using `structs` with function pointer interfaces is also a valid method of run-time abstraction [76]. Compile-time abstraction in C++ typically happens through C++ templates [77], though C preprocessor macros can also be used to supply some compile-time polymorphism.

Methods of implementing abstraction layers at runtime necessarily introduce some amount of overhead involved in function calls expressed in the API. These API mechanisms require dynamic function dispatching which introduces overhead [78]. There is no viable method to reduce a runtime abstraction layer overhead to zero, though the impact is typi-

cally minimal in the general case. In general code development and usage this overhead is negligible enough that the ease of use of most dynamic dispatching mechanisms is a reasonable tradeoff for a small performance hit. Runtime abstractions are very general and provide the most utility and responsiveness to uncertain requirements being placed on the system. In event-based and real-time systems these abstraction layers provide vital flexibility and utility.

Implementing a compile-time abstraction layer allows a developer to leverage the power of an abstract interface while not paying as steep a price for the abstraction. Typically abstraction layers using compile-time abstractions result in the compiled code having only the overhead of a single function call associated with use of functions in the API. However, since the abstraction layer is made concrete at compile-time, the software system cannot respond as well to runtime effects and conditions that may require the software to respond with flexibility. Compile-time abstractions provide the best utility in high-performance or embedded systems where the requirements on the program are constrained and can be determined a priori.

5.4 High Performance Loops

Most HPC software possesses sections that can be referred to as tight-loops. These are low-level loops over simple data structures that represent the bulk of the important computational effort expended in a program. In HPC these include linear system operations (matrix assembly, matrix multiplication, linear system solution), integration loops over discretized domains, etc. Reduction of overhead inside of these loops is an especially important target of program optimization and improvement, as a minor loop speedup has an outsized effect on program speedup due to the sheer number of times the speedup will effect the computation [79].

Further, optimizing these tight loops to take advantage of features such as vectorization [80] and other considerations for tight loops on heterogenous architectures such as supercomputers with on-node accelerators such as CUDA accelerators. Specifically with respect to accelerators such as CUDA cards, branching operations and context switching are particularly impactful on achieving high-performance code in tight loops.

Abstraction layers supplying operations used in tight-loops in an application have a more stringent set of requirements in order to provide true utility to an HPC developer,

see [74] for an examination of the MPI standard in this context.

5.5 Zero Overhead Abstraction

Given the requirements on tight-loop HPC operations for absolutely minimal overhead but also a desire to explore different implementations of key operations and to allow the software to perform well on machines with different hardware configurations, the use of some abstraction layer with absolutely minimal cost is desirable. As HPC programs fall within the set of programs with known constraints at compile-time, the use of a compile-time abstraction layer is the natural choice for tight-loop operations. However, compile-time abstraction layers still typically pay the price of a single function call per API operation.

This can be overcome through aggressive inlining of every abstract function implementation for the API backend, allowing the compiler (in release mode with sufficient optimizations active) to inline the abstract function calls. In C/C++ there is no guarantee by the standard that a function which can be inline will actually be inlined [81], though many compilers offer extensions to force inlining of functions [82]–[84]. Through the use of a compile-time abstraction mechanism and aggressive/forced inlining, the abstraction layer can disappear in the compiled code, resulting in zero-overhead abstraction.

5.5.1 Effect on Vectorization

Of importance is the effect this API model has on the vectorization of tight loops. Vectorization is a process whereby the operations in a loop occur simultaneously to multiple operands over which the loop is occurring. This on-core model of single-instruction multiple-data (SIMD) style parallelism effectively reduces the number of loop cycles necessary to process the operands, massively reducing compute time to complete the loop. Vectorization and the code requirements to fully take advantage of vectorization are complex issues that vary from compiler to compiler, as each offers differing mechanisms and analysis algorithms to vectorize loops. Further discussion of vectorization pertaining specifically to HPC can be found in [85].

Vectorization of loops containing function calls is a more recent additional the capabilities of vectorization routines in modern compilers [86], and in order to vectorize a loop containing a function call, certain information must be known about the function at compile-time. Further, dynamic function dispatch methods make supporting vectorization

Table 5.1: LAS Zero-overhead API

Operation	Description
`zero(Mat*)`	zero a matrix
`zero(Vec*)`	zero a vector
`zero(Mat*,int)`	zero a row in a matrix
`add(Mat*,int,int*,int,int*,double*)`	add values to locations in a matrix
`add(Vec*,int,int*,double*)`	add values to locations in a vector
`set(Mat*,int,int*,int,int*,double*)`	set values at locations in a matrix
`set(Vec*,int,int*,double*)`	set values at locations in a vector
`get(Mat*,int,int*,int,int*,double**)`	get values from locations in a matrix
`get(Vec*,int,int*,double**)`	get values from locations in a vector
`norm(Vec*)`	calculate the 2-norm of a vector
`dot(Vec*,Vec*)`	calculate the dot-product of two vectors
`axpy(double,Vec*,Vec*`	$y = ax + y$ operation on two vectors
`get(Vec*,double**)`	extract the entire local vector array
`restore(Vec*,double*)`	restore the entire local vector array

very difficult only a single compiler (the Intel C++ compiler after version 17.0) has (limited) support for vectorizing loops containing C++ virtual function calls [87].

Adhering to the strict restrictions and implementation specifics to take advantage of loop vectorization is a difficult programming problem. Using a zero-overhead abstraction layer as specified in section 5.5 to inline function operations has the additional benefit of reducing the complexity of work required to refactor existing loops to fully take advantage of the benefits of vectorization, so long as the backend implementation itself allows for vectorization.

5.6 Linear Algebraic System (LAS)

The Linear Algebraic System (LAS) [88] is a zero-overhead abstraction layer library for performing tight-loop operations on linear algebra data structures in HPC applications. The library provides a zero-overhead API for manipulating matrices and vectors, as well as a set of run-time APIs for performing operations which themselves abstract tight-loop operations such as solving a linear system, matrix-vector multiplication, and matrix-matrix multiplication. The library supports five backends including the 'sparse' and 'core' default backends, Sparskit, a serial LU solver written in Fortran, CuSparse, the sparse linear algebra library in the CUDA toolkit, and PETSc, an MPI-parallel linear algebera package used to construct and solve large linear systems in parallel.

Table 5.2: LAS Nonzero-overhead API

Operation	Description
`Mat * create()`	create a matrix
`Vec * create()`	create a vector
`LasOps * getLasOps<T>()`	retrieve API for backend `T`
`solve(Mat*,Vec*,Vec*)`	solve a linear system $Ax = b$
`mult(Mat*,Vec*)`	perform a matrix-vector multiply
`mult(Mat*,Mat*)`	perform a matrix-matrix multiply

The set of operations included in the LAS zero-overhead API (and a description of each) is listed in table 5.1, and the set of operations included in the extended API (using standard C++ inheritance polymorphism) is listed in table 5.2.

5.6.1 Implementation using CRTP

The LAS abstraction layer makes use of the curiously-recurring template pattern (CRTP) in C++ to provide compile-time abstraction. An example of the use of this C++ design pattern in the LAS library from the interface specification to the backend implementation is shown in listing 5.1. The primary interface class `las::LasOps<C>` declared in the `las.h` header defines the zero-overhead API of functions. In the minimal example in the listing only a single function – that for zeroing an opaque matrix type – is exposed. The implementation of the `zero` function is defined inline in the class declaration and consists only of performing a compile-time cast of the object to the template type and calling an equivalent, unexposed version of the API defined in the template class `_zero`.

Listing 5.1: LAS Implementation Using CRTP

```
1  // las.h
2
3  // opague matrix type
4  class Mat;
5
6  // interface definition using CRTP
7  template <class C>
8  class LasOps<C>
9  {
10 public:
11   void zero(Mat * m)
```

```
12    {
13      static_cast<C*>(this)->_zero(m);
14    }
15  };
16
17  // function to retrieve interface objects
18  //  for specific backends
19  template <class C>
20  LasOps<C> * getLasOps();
```

This is the core mechanism enabled by the CRTP pattern for compile-time abstraction. Since the type `C` is known at compile time, the subsequent location of the call to the `C::_zero` function is also known, and can be inlined as well. In this way the invocation of the `LasOps::zero()` function can be replaced at compile time be the concrete backend implementation `C::_zero()`. Further, a templated function to retrieve a version of the `las::LasOps<C>` for a particular backend `C` is also defined.

5.6.2 Backend Implementation

To implement a backend for the API, a pair of header files is required. Listings 5.2 and 5.3 show the implementation of the `zero(Mat*)` function for the `sparse` backend. The declaration of an opaque class `sparse` used as a template parameter and the template specialization of the function for retrieval of the interface-exposing `LasOps<sparse>` structure is the entire public interface required to make use of the API for the `sparse` backend.

Listing 5.2: LAS Backend Public Interface

```
1  // lasSparse.h  public interface for 'sparse' backend
2  // opaque class for template specialization for this backend
3  class sparse;
4  // specialization of the API retrieval function
5  template<>
6  LasOps<sparse> * getLasOps<sparse>();
7  // include the implementation header
8  #include "lasSparse_impl.h"
```

All implementation details and explicit calls to any dependencies used by the backend are placed in an `xxx_impl.h` file, as the functions are designed to be inlined they compile into the user code rather than being defined in a source file.

The `sparse_impl.h` file in listing 5.3 defines data structures used in the backend. In order to implement the `LasOps<sparse>::_zero(Mat*)` function the `csrMat` and `CSR` types are needed. Additionally the `LasOps<sparse>` class is declared and the hidden API function `_zero(Mat*)` is implemented inline. The combination of listings 5.1, 5.2, and 5.3 define the entire API and `sparse` backend for the `zero(Mat*)` function.

Listing 5.3: LAS Backend Implementation

```
1  // lasSparse_impl.h  implementation of the 'sparse' backend
2  // compressed sparse row format matrix,
3  //  sparsity pattern and nonzero values
4  //  are handled separately
5  class csrMat
6  {
7  private:
8    CSR * csr;
9    double * nonzeros;
10 public:
11   csrMat(CSR * c)
12     : csr(c)
13     , nonzeros(new double[csr->getNNZ()])
14   { }
15   void zero()
16   {
17     memset(&nonzeros[0],0,sizeof(double)*csr->getNNZ());;
18   }
19 }
20
21 // implementation of API operations of the 'sparse' backend
22 class SparseOps : public LasOps<SparseOps>
23 {
24 public:
25   void _zero(Mat * m)
26   {
```

```
27      csrMat * cm = reinterpret_cast<csrMat*>(m);
28      cm->zero();
29    }
30  };
31
32  // specialized function definition to retrieve
33  //  API object.
34  template <>
35  LAS_INLINE LasOps<sparse> * getLasOps<sparse>()
36  {
37    static SparseOps * ops = nullptr;
38    if(ops == nullptr)
39      ops = new SparseOps;
40    return ops;
41  }
```

Two default backends are provided with LAS, the `sparse` backend which operates on CSR format matrices [89] and the `dense` backend which operates on logically 2d arrays as matrices. The CSR matrices from the `sparse` backend provide additional utility as they can be used directly with additional solver/multiplication backends without modification.

5.7 Automated C API Generation

Leveraging the full capabilities of the zero-overhead API implemented in LAS requires the calling language be C++. However C and FORTRAN are still widely used in high performance computing, as many existing pieces of software exist only in these languages. In order to provide utility to such existing codebases, it is common to provide a C API with a library, as FORTRAN and C have well-established mechanisms (especially `iso_c_binding` since FORTRAN90) to interoperate, so all C and FORTRAN codebases can make use of the C API.

Each backend added to the LAS library automatically generates a C API interface allowing it to be used from these existing codebases. In order to allow different backends to be used in the same link unit in a C program, the C API generate by two different backends cannot define the same symbols. This is accomplished by generating custom function names at compile time for each function in the zero-overhead API, through the use of a preprocessor macro automatically created during the CMake configuration process for each backend. This

macro – `las()` – is used in a template `las_capi.h` file to generate a backend-specific C API file, e.g. `las_sparse_capi.h` for the `sparse` backend. See listing 5.4 for the full macro definition for the `sparse` backend.

Listing 5.4: LAS C API Macro

```
1 #define CONCAT_FUNC(pkg,fnc,bck) pkg ## _ ## fnc ## _ ## bck
2 #define MAKE_PKG_FUNC(pkg,fnc,bck) CONCAT_FUNC(pkg,fnc,bck)
3 #define las(FNC) MAKE_PKG_FUNC(las,FNC,BACKEND)
4 #define BACKEND sparse
```

Listing 5.5 shows the usage of the macro at configuration and compile time to automatically create a C API for the `sparse` backend first discussed in section 5.6.2.

Listing 5.5: LAS C API Macro Usage

```
1 // las_capi.h
2 // c api to zero a matrix for a generic backend
3 void las(zero_mat)(las_ops * las, las_mat * mat);
4
5 // las_sparse_capi.h
6 // auto-generated api macro results in
7 //  las(zero_mat) expanding to las_zero_mat_sparse
8 #define BACKEND sparse
9 void las(zero_mat)(las_ops * las, las_mat * mat);
```

Since the C API for each backend generates different symbols for the API functions, we are able to retain the benefit of configuration and compile-time polymorphism. Using the `las()` macro in conjunction with redefining the `BACKEND` macro either directly in code or using placeholders replaced at configuration time by the CMake configuration system, multiple backends can be used (and determined at compile-time) in any single compilation unit.

5.8 LAS Adoption and Usage

LAS is currently used in several HPC simulation codes. First and foremost it is used in the soft biological tissue code discussed in chapter 3, where it was initially developed.

It is used in the biotissue code in the micro-scale RVE fiber network simulation to create and assemble the global Jacobian and force vector and to solve the resulting linear system $Ku = -f$. The Sparskit [53] linear system solver and 'sparse' LAS backend which operates on and enables the construction of CSR matrices and standard arrays as vectors (as seen in the `csrMat` implementation in listing 5.3) is used to initialize, assemble and solve the micro-scale problem in serial.

However, when using sufficiently large problem sizes at the micro-scale (see section 6.2.2.1) the solution of a single RVE will need to be parallelized. By using LAS non of the finite element assembly operations on the linear system will need to be rewritten to accommodate this parallelization, only the configuration of a different backend (such as the existing PETSc [23]–[25] backend is required at configuration and compilation time.

In the XGCM PIC fusion code [90], [91] it is being used to construct matrices used to perform particle-to-field deposition and field recovery operations which interpolate tensor field values from nodes on a discrete mesh to values on discrete particles in the simulation and vice-versa. The LAS code was initially developed to target this application as it is intended to be deployed on a variety of HPC target platforms with varying hardware profiles.

An LAS backend using the cuSparse [92] library for sparse matrix operations as part of the the CUDA [93] toolkit was implemented for use in this operation on machines providing CUDA-enabled computational accelerators. Additionally a Kokkos [94] backend is being implemented for use on multi-core machines without accelerators.

The M3D-C1 code is a fusion code used to simulate the macroscopic state of the plasma inside a tokamak fusion ring. Specifically this code focuses on the effects of the interior boundary surfaces on the magnetic fields and plasma state through the direct modeling and inclusion in the simulation domain of the wall boundary geometry [95], [96]. The LAS library configured with a PETSc backend is currently being used to support the linear system operations for the simulation, which consists of approximately 2 dozen separate matrices.

Many of these matrices are only used locally on a process for multiplication operations, which may benefit from a switch away from PETSc to another backend more specialized for the target architecture. Using LAS for this codebase will allows targeted use of specific backends for specific hardware configurations to achieve the most computational benefit, while retaining the benefits of PETSc for larger parallel matrices.

5.9 Zero Overhead API Evaluation

A test binary in which a parallel partitioned-mesh finite element problem is formulated and the nonzero structure of the global matrix is pre-calculated and pre-allocated was developed using the SCOREC/core libraries. The PETSc linear algebra package is used to create and manage the global parallel matrix. In the elemental numerical integration routines a call is made to assemble the elemental linear system into the global linear system using a global DOF numbering scheme. This call to assemble the elemental systems is replaced in various versions of the code with different API mechanisms.

The lowest possible overhead method of using the PETSc API is simply a direct function call to the `MatSetValues()` command, all abstraction layers and calling conventions will eventually make a call to this function to assemble values into an (unblocked) PETSc matrix.

Secondly, the `LasOps<petsc>` backend is used for the assembly process.

Third, a single call to a function in a separate compilation unit (rather than an inlined function, which would compile down to simply the PETSc API call) is used. While it is possible to implement an API with compile-time configurable backends in this manner it would either require the use of a configuration mechanism like the preprocessor macro in section 5.7 to generate unique link symbols or only a single backend could be used in a binary link-scope as the symbols for different backends would conflict.

Forth, a C-style virtual function API is used where a structure consisting of function pointers is created and they are bound at run time to the correct function.

Finally, the fifth method is C++ inheritance polymorphism, where an object API is defined and an inheriting object defines that API; the main compilation unit where the API is used is not aware of the inherited type, only the supertype.

5.9.1 Assembly Analysis

The LAS library was configured, compiled, and linked on the NERSC CORI Cray XC40, using the Intel compiler suite (version 18.0.1) with compilation and link flags for the `O3` optimization level and using the `c++11` standard. Additionally the library was built on the the AMOS BlueGene/Q computer, using the IBM XL compiler suite (version 12.0) again with the `O3` optimization level and the `c++11` standard.

The assembly of the resulting linked binary was extracted for the section of code seen in

listing 5.6 for each test case noted in section 5.9. The abbreviated assembly code generated for both the call to the function and any function backend is included in the listings in table 5.4 for the PowerPC assembly generated by the XL compiler suite on the AMOS BG/Q and in table 5.3 for the x86-64 assembly generated by the Intel compiler suite.

Listing 5.6: LAS Test Code

```
1  #if defined(RAW)
2    MatSetValues(K,nds_per_lmt,&nums[0],nds_per_lmt,&nums[0],&ke[0],
       ADD_VALUES);
3  #elif defined(LAS)
4    las_ops->assemble(las_K,nds_per_lmt,&nums[0],nds_per_lmt,&nums[0],&ke
       [0]);
5  #elif defined(CALL)
6    add(K,nds_per_lmt,&nums[0],nds_per_lmt,&nums[0],&ke[0]);
7  #elif defined(CVIRT)
8    (*petsc_cops->add)(K,nds_per_lmt,&nums[0],nds_per_lmt,&nums[0],&ke[0]);
9  #elif defined(VIRT)
10   petsc_ops->add(las_K,nds_per_lmt,&nums[0],nds_per_lmt,&nums[0],&ke[0]);
11 #endif
```

Table 5.3: CORI Cray XC40 x86-64 Assembly for API Calling Methodologies

Function call (1):

```
vmovsd %xmm0,(%rsp)
mov    0x28(%rsp),%rdx
add
    $0xfffffffffffffff0
    ,%rsp
mov    %r12d,%esi
mov    %r12d,%ecx
mov    %rdx,%r8
mov    0x60(%rsp),%rdi
mov    0x18(%rsp),%r9
movl   $0x2,(%rsp)
callq  767ee0 <
    MatSetValues>
```

LasOps<petsc>
function call (2):

```
vmovsd %xmm0,(%rsp)
mov    0x28(%rsp),%rdx
add
    $0xfffffffffffffff0
    ,%rsp
mov    %r12d,%esi
mov    %r12d,%ecx
mov    %rdx,%r8
mov    0x60(%rsp),%rdi
mov    0x18(%rsp),%r9
movl   $0x2,(%rsp)
callq  767f00 <
    MatSetValues>
```

Non-inlined
function call (3):

```
vmovsd %xmm0,(%rsp)
mov    0x28(%rsp),%rdx
mov    %r12d,%esi
mov    %r12d,%ecx
mov    %rdx,%r8
mov    0x50(%rsp),%rdi
mov    0x8(%rsp),%r9
callq  4135e0 <add()>

<add()>:
 sub    $0x18,%rsp
 movl   $0x2,(%rsp)
 callq  767ed0 <
    MatSetValues>
 add    $0x18,%rsp
 retq
```

C-Style
Polymorphism (4):

```
vmovsd %xmm0,(%rsp)
mov    0x28(%rsp),%rdx
mov    %r12d,%esi
mov    0x50(%rsp),%rdi
mov    %r12d,%ecx
mov    0x8(%rsp),%r9
mov    %rdx,%r8
callq  *0x0(%r13)

<_add()>:
sub    $0x18,%rsp
movl   $0x2,(%rsp)
callq  767ed0 <
    MatSetValues>
add    $0x18,%rsp
retq
```

C++ Inheritance
Polymorphism (5):

```
vmovsd %xmm0,(%rsp)
mov    0x28(%rsp),%rcx
add
    $0xfffffffffffffff0
    ,%rsp
mov    0x0(%r13),%r10
mov    %r13,%rdi
mov    0x18(%rsp),%rax
lea    0x60(%rsp),%rsi
mov    %r12d,%edx
ov    %r12d,%r8d
mov    %rcx,%r9
mov    %rax,(%rsp)
callq  *(%r10)

<petsc_ops::add()>:
mov    0x8(%rsp),%rax
mov    (%rsi),%rdi
mov    %edx,%esi
mov    %rcx,%rdx
mov    %r8d,%ecx
mov    %r9,%r8
mov    %rax,%r9
movl   $0x2,0x8(%rsp)
jmpq   767ed0 <
    MatSetValues>
nopl   0x0(%rax,%rax,1)
nopl   0x0(%rax,%rax,1)
retq
```

Table 5.4: AMOS BG/Q PowerPC Assembly for API Calling Methodologies

Function call (1):

```
ld  r5,168(r1)
ld  r3,328(r1)
ld  r8,192(r1)
mr  r4,r14
mr  r6,r14
li  r9,2
mr  r7,r5
bl  0dc8e0 <.
    MatSetValues>
```

LasOps<petsc>
function call (2):

```
ld  r5,168(r1)
ld  r3,0(r21)
ld  r8,192(r1)
mr  r4,r17
mr  r6,r17
li  r9,2
mr  r7,r5
bl  10dc910 <.
    MatSetValues>
```

Non-inlined
function call (3):

```
ld    r5,168(r1)
ld    r3,328(r1)
ld    r8,192(r1)
mr    r4,r14
mr    r6,r14
mr    r7,r5
bl    1002160<.add>

<.add>:
stdu  r1,-128(r1)
mflr  r0
std   r0,144(r1)
extsw r4,r4
extsw r6,r6
li    r9,2
bl    10dc8e0<.
    MatSetValues>
nop
ld    r12,144(r1)
addi  r1,r1,128
mtlr  r12
blr
```

C-Style
Polymorphism (4):

```
ld    r9,0(r21)
ld    r5,168(r1)
ld    r3,328(r1)
ld    r8,192(r1)
mr    r4,r14
mr    r6,r14
ld    r0,0(r9)
mr    r7,r5
ld    r11,16(r9)
mtctr r0
ld    r2,8(r9)
bctrl

<._add()>:
stdu  r1,-128(r1)
mflr  r0
std   r0,144(r1)
extsw r4,r4
extsw r6,r6
li    r9,2
bl    10dc8f0<.
    MatSetValues>
nop
ld    r12,144(r1)
addi  r1,r1,128
mtlr  r12
blr
```

C++ Inheritance
Polymorphism (5):

```
ld    r7,0(r20)
ld    r6,168(r1)
mr    r5,r14
ld    r9,192(r1)
mr    r3,r20
mr    r4,r21
ld    r10,0(r7)
mr    r7,r14
mr    r8,r6
ld    r0,0(r10)
ld    r11,16(r10)
mtctr r0
ld    r2,8(r10)
bctrl

<.petsc_ops::add()>:
stdu  r1,-128(r1)
mflr  r0
std   r0,144(r1)
extsw r0,r7
ld    r3,0(r4)
extsw r4,r5
mr    r5,r6
mr    r6,r0
mr    r7,r8
mr    r8,r9
li    r9,2
bl    10dc900<.
    MatSetValues>
nop
ld    r12,144(r1)
addi  r1,r1,128
mtlr  r12
blr
```

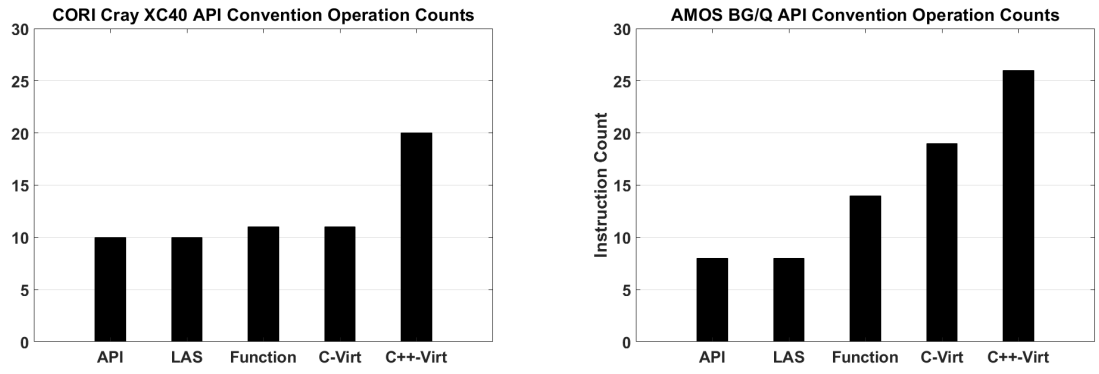

Fig. 5.1: Assembly Operation Counts for API Calling Methods

For case 1 – the explicit PETSc API function call – and case 2 – the LAS API configured at compile-time with PETSc as the backend – compile down to identical instructions in both test environments. In case 3 – a standard function call – the assembly includes a branch or call instruction to switch contexts to the `add()` function before the PETSc call is prepared and made. In case 4 – C-style inheritance polymorphism – the branch/call location is first calculated prior to being invoked, at which point the PETSc API call is prepared and made. In case 5 – C++ inheritance polymorphism – the branch/call location is again calculated, this time from the virtual function table of the API object, before the branch occurs and the PETSc call is prepared and made.

Figure 5.1 shows the assembly operation count for each case on both target architectures. In both cases the raw API call and the LAS API calls are identical. On the CORI Cray XC40 the function call and C-style virtual function calls take the same number of operations, only a single additional operation over the raw API or LAS API. On the AMOS BG/Q the function call requires almost twice the number of operations, and the C-style virtual function call requires around 2.5 times as many. In both cases the C++ virtual function call requires the most assembly instructions, on CORI it requires double the number of instructions as the raw API or LAS API, on AMOS it requires 3 times as many.

5.10 Conclusion

In this work we have developed a methodology for implementing zero-overhead abstraction layers for using in HPC environments. We explored the utility of abstraction layers in general and in the specific context of HPC computing, further we discussed different

paradigms of abstraction layer implementation and their impact on utility and performance. We determined that for HPC implementations focused primarily on performance and correctness the use of a compile-time abstraction layer provides for the best outcomes. We developed a method of implementing compile-time abstraction layers – using C++ to implement the Curiously Recurring Template Pattern (CRTP) – which are compiled away to expose raw calls to third-party libraries over which they are implemented. We discussed the implementation of the Linear Algebraic Systems (LAS) library which is implemented using the methodology developed. We further developed methods to allow the hot-swapping of the LAS abstraction layer backend at configuration time and explored options to provide more accessible C and FORTRAN APIs to allow incorporation of the API in a wider variety of HPC codes with minimal overhead. We briefly discussed the existing applications in which LAS is used including a multi-scale simulation of soft biological tissues and two PIC fusion codes. Finally we established the validity of our approach by analyzing a small API usage example using several API conventions and showing that on different platforms and using different compilers, the assembly code produced by our approach is identical to that produced by raw calls to an API backend, while every other approach introduces additional assembly operations and overhead.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

In this work we have discussed the development of mechanisms and software to support the implementation of multi-scale software on HPC machines from existing single-scale simulations, the use of our software to implement a multi-scale simulation of soft tissue, the extension of our methods to allow for the dynamic load balancing of individual scale in a multi-scale system, and finally the design and implementation of zero-overhead abstraction layers used in the several HPC codes including the soft tissue simulation.

In chapter 2 we began by discussing the difficulties and motivations in the construction of multi-scale simulations. We developed an abstract model of multi-scale coupling which is congruent with existing multi-scale abstractions. We developed a software infrastructure, AMSI to support the implementation of multi-scale systems from existing SPMD HPC simulations in agreement with the multi-scale abstract model. The implications of the design and implementation details of AMSI were discussed. These developments make possible the creation of novel new multi-scale simulations and couplings without tremendous development costs associated with the development of bespoke codes.

In chapter 3 we further used the AMSI software to combine two single-scale simulations together to produce a multi-scale simulation of soft biological tissues. We discussed the properties of each simulation scale and then described the implementation details of each scale and how AMSI is used to couple the two scales together into a multi-scale system. The application of the multi-scale soft tissue code to several problems of interest was discussed including a standard tensile test sample domain, a model of a glomular podocyte cell, and a single neuron cell embedded in a collagen gel. The numerical and parallel characteristics of the soft tissue simulation code were discussed.

In chapter 4 we extended our work on AMSI to allow for the dynamic movement of data between processes assigned to individual scales in a multi-scale system. We developed techniques to retain the validity of the multi-scale couplings during such operations, and used these capabilities to implement and enact scale-sensitive load balancing operations on the multi-scale soft tissue code discussed in chapter 3.

In chapter 5 we developed an approach for the implementation of zero-overhead abstraction layers for use in HPC environments. We showed the application of this approach to the implementation of the Linear Algebraic Systems (LAS), which is currently used in several HPC application codes. The implementation approach was extended to auto-generate addition APIs to allow C and FORTRAN users to access the LAS backends with minimal overhead. Finally the approach was validated through the evaluation of several interface implementation mechanisms on two HPC platforms using two proprietary compilers.

6.2 Future Work

6.2.1 AMSI Extension to Concurrent Multi-scale Problems

The AMSI multi-scale capabilities are currently designed around applications to hierarchical multi-scale simulations. The geometric inter-scale relationship characterization for concurrent multi-scale couplings is significantly more complicated than that for hierarchical multi-scale couplings. Supporting concurrent multi-scale problems in AMSI will require the development and use of geometric relationship couplings. Disjoint simulation scales require an interface through which to establish the extent of their simulated geometric domains, and AMSI must use the information provided to produce a geometric coupling over which an AMSI multi-scale coupling operation can be established

6.2.2 Biotissue Software Extensions

6.2.2.1 Parallelization of the RVE

The original implementation of the micro-scale simulation was developed as a solely serial simulation, with no consideration of parallelization. For RVE systems with hundred or thousands of fibers, the required computational cost is minimal enough that use of message-passing parallelism would introduce sufficient overhead that any computation gain would be lost.

The micro-scale simulation has been rewritten from the ground-up using modern techniques and libraries which will more easily support the execution of parallel simulations on the RVEs. As the fiber network sizes scale the computational demand will pass the threshold where parallelization will increase time-to-solution, and parallel RVEs should be used.

This extension will also require accommodation in the algorithms used to define the AMSI coupling for the biotissue problem.

6.2.2.2 Embedded Fiber Network RVE

A micro-scale RVE model consisting of a fiber network embedded in a nonfibrillar matrix was developed in [97]. The computational expense of these simulations is sufficient to require a parallel implementation, in contrast to the relatively small fiber-only micro-scale RVE simulation used in the current biotissue problem cases. Once the parallelization of the fiber-only RVE simulations discussed in section 6.2.2.1 is complete, the implementation will be extended to include the additional terms required to represent the fiber and nonfibrillar matrix RVE version.

6.2.3 Scale-Sensitive Load Balancing with Over-decomposition and Work Pooling

As noted in section 4.5 an additional approach to load balancing is required in the context of the multi-scale biotissue simulation. Point-to-point load balancing operations requiring weighting persistence to achieve effective load balance over the course of a simulation are insufficient to improve the run-time of the biotissue simulation, due to the lack of persistence in the Newton iteration count of the micro-scale RVE simulations.

An additional layer of work decomposition could allow effective load balancing, where groups of micro-scale processes are assigned pools of micro-scale RVE simulations to work on and process these work-pools each macro-scale Newton-iteration. Rather than load-balancing the individual processes on the micro-scale, the work-pools assigned to micro-scale groups are load balanced. We hypothesize that the work-pooling will result in the workgroups achieving a balanced time-to-solution for the assigned RVEs, while load-balancing on the over-decomposed set of workgroups will migrate the most heavily weighted RVEs in order to balance them among the workgroups in order to reduce differences in time-to-solution between the workgroups set.

6.2.4 LAS Zero-overhead API Usage

The LAS library is currently used in three simulation codes, but has yet to be evaluated in terms of general utility. As it is being used in codes intended to be deployed on different platforms – and with the use of different backends – the efficiency and utility of the LAS abstraction layer needs to be evaluated. While we have shown that the abstraction layer acts as intended from the perspective of the configuration and compilation stages, we intend

further to show the utility of the abstraction layer by allowing and evaluating hot-swapping of backends on target machines to evaluate operational performance. This will require the incorporation of additional backends for the abstraction layer as it currently provides an insufficient variety to provide a broad enough perspective on the efficacy of the approach.

REFERENCES

[1] B. FrantzDale, S. J. Plimpton, and M. S. Shephard, "Software components for parallel multiscale simulation: an example with lammps," *Eng. with Comput.*, vol. 26, no. 2, pp. 205–211, 2010.

[2] M. S. Shephard, E. S. Seol, and B. FrantzDale, "Toward a multi-model hierarchy to support multiscale simulations," *Handbook of Dynamic Sys. Modeling*, vol. 12, pp. 1–18, 2007.

[3] A. Toselli and O. B. Widlund, *Domain Decomposition Methods: Algorithms and Theory*. New York City: Springer, 2005, vol. 34.

[4] R. E. Miller and E. B. Tadmor, "The quasicontinuum method: Overview, applications and current directions," *J. of Computer-Aided Materials Design*, vol. 9, no. 3, pp. 203–239, 2002.

[5] E. Weinan, *Principles of Multiscale Modeling*. Cambridge: Cambridge University Press, 2011.

[6] M. J. Buehler, "Concurrent scale coupling techniques: From nano to macro," MIT CEE, Cambridge, 2006.

[7] J. Fish, *Multiscale Methods: Bridging the Scales in Science and Engineering*. Oxford: Oxford University Press on Demand, 2010.

[8] E. Weinan *et al.*, "The heterognous multiscale methods," *Commun. Math. Sciences*, vol. 1, no. 1, pp. 87–132, 2003.

[9] E. Weinan, B. Engquist, X. Li, W. Ren, and E. Vanden-Eijnden, "Heterogeneous multiscale methods: a review," *Commun. Comput. Phys*, vol. 2, no. 3, pp. 367–450, 2007.

[10] J. Borgdorff *et al.*, "Foundations of distributed multiscale computing: Formalization, specification, and analysis," *J. Parallel Distrib. Comput.*, vol. 73, no. 4, pp. 465–483, 2013.

[11] E. B. Tadmor and R. E. Miller, *Modeling Materials: Continuum, Atomistic and Multiscale Techniques*. Cambridge: Cambridge University Press, 2011.

[12] D. E. Keyes *et al.*, "Multiphysics simulations challenges and opportunities," *Int. J. High Performance Comput. Applications*, vol. 27, no. 1, pp. 4–83, 2013.

[13] M. Berzins *et al.*, "Uintah: a scalable framework for hazard analysis," in *Proc. of the 2010 TeraGrid Conf.*, 2010, p. 3.

[14] ——, "Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM J. Scientific Comput.*, 2015.

[15] W. Joppich and M. Kürschner, "Mpccia tool for the simulation of coupled applications," *Concurrency and Comput.: Practice and Experience*, vol. 18, no. 2, pp. 183–192, 2006.

[16] F. Delalondre, C. Smith, and M. S. Shephard, "Collaborative software infrastructure for adaptive multiple model simulation," *Comput. Methods Appl. Mech. Eng.*, vol. 199, no. 21, pp. 1352–1370, 2010.

[17] The Fraunhofer Institute. *MpCCI Web Page* [Online]. Available: http://www.mpcci.de/, Accessed on: 2018-04-01.

[18] M. B. Belgacem and B. Chopard, "Muscle-hpc: A new high performance api to couple multiscale parallel applications," *Future Generation Comput. Syst.*, vol. 67, pp. 72–82, 2017.

[19] J. Borgdorff *et al.*, "Distributed multiscale computing with muscle 2, the multiscale coupling library and environment," *J. Comput. Science*, vol. 5, no. 5, pp. 719–731, 2014.

[20] ——, "A distributed multiscale computation of a tightly coupled model using the multiscale modeling language," *Procedia Comput. Science*, vol. 9, pp. 596–605, 2012.

[21] D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandie, "Moose: A parallel computational framework for coupled systems of nonlinear equations," *Nucl. Eng. and Design*, vol. 239, no. 10, pp. 1768–1778, 2009.

[22] D. Gaston *et al.*, "Continuous integration for concurrent computational framework and application development," *J. of Open Res. Software*, vol. 2, no. 1, 2014.

[23] S. Balay *et al. PETSc Web page* [Online]. Available: http://www.mcs.anl.gov/petsc, Accessed on: 2018-05-14.

[24] ——, "PETSc users manual," Argonne National Laboratory, Lemont, Tech. Rep. ANL-95/11 - Revision 3.6, 2015, [Online]. Available: http://www.mcs.anl.gov/petsc, Accessed on: 2018-03-06.

[25] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Basel, Switzerland: Birkhäuser Press, 1997, pp. 163–202.

[26] M. Heroux *et al.*, "An Overview of Trilinos," Sandia National Laboratories, Albuquerque, Tech. Rep. SAND2003-2927, 2003.

[27] M. A. Heroux, J. M. Willenbring, and R. Heaphy, "Trilinos Developers Guide," Sandia National Laboratories, Albuquerque, Tech. Rep. SAND2003-1898, 2003.

[28] ——, "Trilinos Developers Guide Part II: ASCI Software Quality Engineering Practices Version 1.0," Sandia National Laboratories, Albuquerque, Tech. Rep. SAND2003-1899, 2003.

[29] M. A. Heroux and J. M. Willenbring, "Trilinos Users Guide," Sandia National Laboratories, Albuquerque, Tech. Rep. SAND2003-2952, 2003.

[30] M. A. Heroux *et al.*, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[31] M. Sala, M. A. Heroux, and D. M. Day, "Trilinos Tutorial," Sandia National Laboratories, Albuquerque, Tech. Rep. SAND2004-2189, 2004.

[32] W. R. Tobin. *The Adaptive Multi-scale Simulation Infrastructure* [Online]. Available: https://github.com/tobinw/amsi, Accessed on: 2018-06-28.

[33] F. Liu and M. Sosonkina, "A multilevel parallelism support for multi-physics coupling," *Procedia Comput. Science*, vol. 4, pp. 261–270, 2011.

[34] Q. Meng, A. Humphrey, and M. Berzins, "The uintah framework: A unified heterogeneous task scheduling and runtime system," *Digital Proceedings of Supercomputing*, 2012.

[35] S. J. Plimpton and J. D. Gale, "Developing community codes for materials modeling," *Current Opinion in Solid State and Materials Science*, vol. 17, no. 6, pp. 271–276, 2013.

[36] A. Hoekstra, B. Chopard, and P. Coveney, "Multiscale modelling and simulation: a position paper," *Phil. Trans. R. Soc. A*, vol. 372, no. 2021, p. 20130377, 2014.

[37] V. Gravemeier, S. Lenz, and W. A. Wall, "Towards a taxonomy for multiscale methods in computational mechanics: Building blocks of existing methods," *Comput. Mech.*, vol. 41, no. 2, pp. 279–291, 2008.

[38] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Comput. Phys.*, vol. 53, no. 3, pp. 484–512, 1984.

[39] R. Löhner, "An adaptive finite element scheme for transient problems in cfd," *Comput. Methods Appli. Mech. Eng.*, vol. 61, no. 3, pp. 323–338, 1987.

[40] F. Broquedis *et al.*, "hwloc: A generic framework for managing fardware affinities in hpc applications," in *Parallel, Distributed and Network-Based Process. (PDP), 2010 18th Euromicro Int. Conf. on*, 2010, pp. 180–186.

[41] W. R. Tobin, J. Merson, V. W. Chan, and D. Fovargue. *Biotissue* [Online]. Available: https://github.com/tobinw/biotissue, Accessed on: 2018-06-28.

[42] Chan, Victor W. L. and others, "Image-based Multi-scale Mechanical Analysis of Strain Amplification in Neurons Embedded in Collagen Gel," *Comput. Methods in Biomechanics and Biomedical Eng.*, in submission.

[43] Simmetrix Incorporated. *Simmetrix Incorporated Web Page* [Online]. Available: http://www.simmetrix.com/, Accessed on: 2018-05-14.

[44] T. Stylianopoulos, "Multiscale mechanical modeling of soft biological tissues," Ph.D. dissertation, Biomedical Eng., University of Minnesota, Minneapolis, MN, 2008.

[45] B. Agoram and V. H. Barocas, "Coupled macroscopic and microscopic scale modeling of fibrillar tissues and tissue equivalents," *J. of Biomedical Eng.*, vol. 123, no. 4, pp. 362–369, 2001.

[46] T. Stylianopoulos and V. H. Barocas, "Multiscale, structure-based modeling for the elastic mechanical fehavior of arterial walls," *J. of Biomechanical Eng.*, vol. 129, no. 4, pp. 611–618, 2007.

[47] X.-J. Luo, T. Stylianopoulos, V. H. Barocas, and M. S. Shephard, "Multiscale computation for bioartificial soft tissues with complex geometries," *Eng. Comput.*, vol. 25, no. 1, p. 87, 2009.

[48] T. Stylianopoulos and V. H. Barocas, "Volume-averaging theory for the study of the mechanics of collagen networks," *Comput. Methods in Appl. Mechanics and Eng.*, vol. 196, no. 31, pp. 2981–2990, 2007.

[49] P. L. Chandran and V. H. Barocas, "Deterministic material-based averaging theory model of collagen gel micromechanics," *J. of Biomechanical Eng.*, vol. 129, no. 2, pp. 137–147, 2007.

[50] X. S. Li, "An overview of superlu: Algorithms, implementation, and user interface," *ACM Trans. Math. Software (TOMS)*, vol. 31, no. 3, pp. 302–325, 2005.

[51] X. S. Li and M. Shao, "A supernodal approach to incomplete LU factorization with partial pivoting," *ACM Trans. Math. Software (TOMS)*, vol. 37, no. 4, 2010.

[52] X. S. Li and J. W. Demmel, "SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Software (TOMS)*, vol. 29, no. 2, pp. 110–140, June 2003.

[53] Y. Saad, "Sparskit: A basic tool kit for sparse matrix computations," Research Inst. for Advanced Computer Science, Moffett Field, Tech. Rep. NASA-CR-185876, May 1990.

[54] J. Dinan, S. Olivier, G. Sabin, J. Prins, P. Sadayappan, and C.-W. Tseng, "Dynamic load balancing of unbalanced computations using message passing," in *Parallel and Distributed Process. Symp., 2007. IPDPS 2007. IEEE Int.*, 2007, pp. 1–8.

[55] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 9, pp. 979–993, 1993.

[56] K. D. Devine *et al.*, "New challenges in dynamic load balancing," *Appl. Numerical Math.*, vol. 52, no. 2, pp. 133–152, 2005.

[57] B. Hendrickson and K. Devine, "Dynamic load balancing in computational mechanics," *Comput. Methods in Appl. Mech. and Eng.*, vol. 184, no. 2, pp. 485–500, 2000.

[58] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. of the 1995 ACM/IEEE Conf. on Supercomputing*, ser. Supercomputing '95, 1995, [Online]. Available: http://doi.acm.org/10.1145/224170.224228, Accessed on: 2018-05-14.

[59] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular fraphs," *SIAM J. Scientific Comput.*, vol. 20, no. 1, pp. 359–392, 1998.

[60] ——, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, 1998.

[61] S. H. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, "Analysis of a graph coloring based distributed load balancing algorithm," *J. of Parallel and Distrib. Computing*, vol. 10, no. 2, pp. 160–166, 1990.

[62] G. Diamond, C. W. Smith, and M. S. Shephard, "Dynamic load balancing of massively parallel unstructured meshes," in *Proc. of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, 2017, p. 9.

[63] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, "Zoltan data management services for parallel dynamic applications," *Comput. in Science and Eng.*, vol. 4, no. 2, pp. 90–97, 2002.

[64] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine, "The zoltan and isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring," *Scientific Programming*, vol. 20, no. 2, pp. 129–150, 2012.

[65] S. Amarasinghe *et al.*, "Exascale software study: Software challenges in extreme scale systems," DARPA IPTO, Air Force Research Labs, Wright-Patterson Air Force Base, OH, Tech. Rep. FA8650-07-C-7724, 2009.

[66] R. Armstrong *et al.*, "Toward a common component architecture for high-performance scientific computing," in *High Performance Distributed Comput., 1999. Proc. The 8th Int. Symp. on*, 1999, pp. 115–124.

[67] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The cca core specification in a distributed memory spmd framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002.

[68] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova, "Overview of gridrpc: A remote procedure call api for grid computing," in *Int. Workshop on Grid Comput.*, 2002, pp. 274–278.

[69] M. Sato, T. Boku, and D. Takahashi, "Omnirpc: a grid rpc system for parallel programming in cluster and grid environment," in *3rd IEEE/ACM Int. Symp. on Cluster Comput. and the Grid, 2003. Proc.*, 2003, pp. 206–213.

[70] J.-L. Falcone, B. Chopard, and A. Hoekstra, "Mml: Towards a multiscale modeling language," *Procedia Comp. Sci.*, vol. 1, no. 1, pp. 819–826, May 2010.

[71] J. Gregersen, P. Gijsbers, and S. Westen, "Openmi: Open modelling interface," *J. of Hydroinformatics*, vol. 9, no. 3, pp. 175–191, 2007.

[72] I. F. Sbalzarini, "Abstractions and middleware for petascale computing and beyond," in *Technol. Integration Advancements in Distributed Syst. and Comput.* IGI Global, 2012, pp. 161–178.

[73] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister, "Software standards for the multicore era," *IEEE Micro*, vol. 29, no. 3, 2009.

[74] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca, "Analysis of the component architecture overhead in open mpi," in *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, 2005, pp. 175–182.

[75] R. Harper and G. Morrisett, "Compiling polymorphism using intensional type analysis," in *Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 1995, pp. 130–141.

[76] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C Programming Language*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[77] S. Haney and J. Crotinger, "How templates enable high-performance scientific computing in c++," *Comput. in Science & Eng.*, vol. 1, no. 4, pp. 66–72, 1999.

[78] R. A. Ballance, A. J. Giancola, G. F. Luger, and T. J. Ross, "A framework-based environment for object-oriented scientific codes," *Scientific Programming*, vol. 2, no. 4, pp. 111–121, 1993.

[79] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Extreme Scaling Workshop (XSW), 2013*, 2013, pp. 18–24.

[80] V. N. Ngo and W.-T. Tsai, "Outer loop vectorization," USA Patent 5,802,375, Sep. 1, 1998.

[81] *Information Technology - Programming Languages - C++*, ISO, Geneva, Switzerland Standard ISO 14 882:2014, 2016.

[82] Stahlman, Richard and others, "Using the GNU Compiler Collection - For Version 7.3.0," GNU Press, Free Software Foundation, Boston, Manual, 2017.

[83] B, Dylan and others, "Intel C++ Compiler User and Reference Guides," March 2018.

[84] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proc. of the Int. Symp. on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004, pp. 75–88.

[85] X. Tian, H. Saito, S. V. Preis, E. N. Garcia, S. S. Kozhukhov, M. Masten *et al.*, "Practical simd vectorization techniques for intel® xeon phi coprocessors," in *Parallel and Distributed Process. Symp. Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th Int.*, 2013, pp. 1149–1158.

[86] T. Xinmin. *[cde-dev] Proposal for Function Vectorization and Loop Vectorization with Function Calls* [Online]. Available: http://lists.llvm.org/pipermail/cfe-dev/2016-March/047732.html, Accessed on: 2018-05-17.

[87] C. Yuan. *Virtual Vector Function Supported in Intel C++ Compiler 17.0* [Online]. Available: https://software.intel.com/en-us/articles/virtual-vector-function-supported-in-intel-c-compiler-170, Accessed on: 2018-05-17.

[88] W. R. Tobin. *Linear Algebraic Systems (LAS)* [Online]. Available: https://github.com/tobinw/las, Accessed on: 2018-06-28.

[89] W. F. Tinney and J. W. Walker, "Direct solutions of sparse network equations by optimally ordered triangular factorization," *Proc. IEEE*, vol. 55, no. 11, pp. 1801–1809, 1967.

[90] S. Ku *et al.*, "Gyrokinetic particle simulation of neoclassical transport in the pedestal/scrape-off region of a tokamak plasma," in *J. of Phys.: Conference Series*, 2006, vol. 46, no. 1, p. 87.

[91] E. DÁzevedo *et al.*, "The fusion code xgc: Enabling kinetic study of multi-scale edge turbulent transport in iter," in *Exascale Scientific Applications: Scalability and Performance Portability*. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2017, p. Chapter 24.

[92] NVIDIA, "Cusparse library," NVIDIA Corporation, Santa Clara, Tech. Rep., 2014.

[93] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," in *ACM SIGGRAPH 2008 classes*, 2008, p. 16.

[94] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. of Parallel and Distributed Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.

[95] N. Ferraro, S. Jardin, L. Lao, M. Shephard, and F. Zhang, "Multi-region approach to free-boundary three-dimensional tokamak equilibria and resistive wall instabilities," *Phys. of Plasmas*, vol. 23, no. 5, p. 056114, 2016.

[96] S. Jardin *et al.*, "The m3d-c1 approach to simulating 3d 2-fluid magnetohydrodynamics in magnetic fusion experiments," in *J. of Phys.: Conference Series*, 2008, vol. 125, no. 1, p. 012044.

[97] L. Zhang, "Microstructural modeling of cross-linked fiber network ambedded in continuous matrix," Ph.D. dissertation, Mechanical Engineering, Rensselaer Polytechnic Institute, Troy, NY, 2013.

[98] J. Bonet and R. D. Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, 2nd ed. Cambridge: Cambridge University Press, Jan. 2008.

[99] V. K. Lai, M. F. Hadi, R. T. Tranquillo, and V. H. Barocas, "A Multiscale Approach to Modeling the Passive Mechanical Contribution of Cells in Tissues," *J. Biomechanical Eng.*, vol. 135, no. 7, p. 071007, Jul. 2013.

[100] P. L. Chandran and V. H. Barocas, "Deterministic Material-Based Averaging Theory Model of Collagen Gel Micromechanics," *J. Biomechanical Eng.*, vol. 129, no. 2, pp. 137–11, 2007.

[101] S. Nemat-Nasser and M. Hori, *Micromechanics: Overall Properties of Heterogeneous Materials*. Amsterdam: Elsevier, Oct. 22, 2013, vol. 37.

APPENDICES

APPENDIX A
Soft Tissue Macro-scale Derivation

A.1 Strain Tensors

The relationship between an elemental vector in the material configuration, $d\boldsymbol{X}$, and the spatial configuration, $d\boldsymbol{x}$ is given by the deformation gradient tensor $\boldsymbol{F}$

$$d\boldsymbol{x} = \boldsymbol{F} d\boldsymbol{X}, \tag{A.1}$$

where $\boldsymbol{F}$ is a two-tensor defined as

$$\boldsymbol{F} = \sum_{i,j=1}^{3} \frac{\partial x_i}{\partial X_j} \boldsymbol{e}_i \otimes \boldsymbol{E}_j = \begin{bmatrix} \frac{\partial x_1}{\partial X_1} & \frac{\partial x_1}{\partial X_2} & \frac{\partial x_1}{\partial X_3} \\ \frac{\partial x_2}{\partial X_1} & \frac{\partial x_2}{\partial X_2} & \frac{\partial x_2}{\partial X_3} \\ \frac{\partial x_3}{\partial X_1} & \frac{\partial x_3}{\partial X_2} & \frac{\partial x_3}{\partial X_3} \end{bmatrix}. \tag{A.2}$$

To measure general deformation, consider two elemental vectors $d\boldsymbol{X}_1$ and $d\boldsymbol{X}_2$ as they deform to $d\boldsymbol{x}_1$ and $d\boldsymbol{x}_2$. Such a deformation will involve stretching and changes in angle between the two elemental vectors. To find strain, which involves only the stretch of the vectors, consider the dot products of elemental vectors

$$\begin{aligned} d\boldsymbol{x}_1 \cdot d\boldsymbol{x}_2 &= (\boldsymbol{F} d\boldsymbol{X}_1) \cdot (\boldsymbol{F} d\boldsymbol{X}_2) \\ &= d\boldsymbol{X}_1 \cdot \boldsymbol{F}^T \boldsymbol{F} d\boldsymbol{X}_2 \\ &= d\boldsymbol{X}_1 \cdot \boldsymbol{C} d\boldsymbol{X}_2, \end{aligned} \tag{A.3}$$

where Eq. (A.1) has been applied and $\boldsymbol{C} \equiv \boldsymbol{F}^T \boldsymbol{F}$ is the right Cauchy-Green deformation

This chapter authored in conjunction with: J. Merson, V. W. L. Chan

tensor (note that $\boldsymbol{F}$ is on right side). One can also express the inverse relationship

$$\begin{aligned} d\boldsymbol{X}_1 \cdot d\boldsymbol{X}_2 &= (\boldsymbol{F}^{-1} d\boldsymbol{x}_1) \cdot (\boldsymbol{F}^{-1} d\boldsymbol{x}_2) \\ &= d\boldsymbol{x}_1 \cdot (\boldsymbol{F}^{-1})^T \boldsymbol{F}^{-1} d\boldsymbol{x}_2 \\ &= d\boldsymbol{x}_1 \cdot \boldsymbol{b}^{-1} d\boldsymbol{x}_2, \end{aligned} \tag{A.4}$$

where $\boldsymbol{b} \equiv \boldsymbol{F}\boldsymbol{F}^T = (\boldsymbol{b}^{-1})^{-1} = ((\boldsymbol{F}^T)^{-1}\boldsymbol{F}^{-1})^{-1}$ is the left Cauchy-Green deformation tensor (note that $\boldsymbol{F}$ is on left side).

The strain can be defined as the change of length from the material to spatial configuration

$$\begin{aligned} \frac{1}{2}\left(d\boldsymbol{x}_1 \cdot d\boldsymbol{x}_2 - d\boldsymbol{X}_1 \cdot d\boldsymbol{X}_2\right) &= \frac{1}{2}\left(d\boldsymbol{X}_1 \cdot \boldsymbol{C} d\boldsymbol{X}_2 - d\boldsymbol{X}_1 \cdot d\boldsymbol{X}_2\right) = d\boldsymbol{X}_1 \cdot \boldsymbol{E} d\boldsymbol{X}_2 \\ &= \frac{1}{2}\left(d\boldsymbol{x}_1 \cdot d\boldsymbol{x}_2 - d\boldsymbol{x}_1 \cdot \boldsymbol{b}^{-1} d\boldsymbol{x}_2\right) = d\boldsymbol{x}_1 \cdot \boldsymbol{e} d\boldsymbol{x}_2, \end{aligned}$$

where

$$\boldsymbol{E} \equiv \frac{1}{2}\left(\boldsymbol{C} - \boldsymbol{I}\right) \quad \text{and} \quad \boldsymbol{e} \equiv \frac{1}{2}\left(\boldsymbol{I} - \boldsymbol{b}^{-1}\right) \tag{A.5}$$

are Lagrangian (or Green) and Eulerian (or Almansi) strain tensors, respectively.

A.2 Elasticity Tensors

In the case when work done by the stresses during deformation only depend on the initial and final configurations, the behavior of the material is called path-independent. Elastic materials that are path-independent are termed hyperelastic. As a consequence of the path-independent behavior, an elastic potential Ψ per unit undeformed volume can be defined as

$$\Psi(\boldsymbol{F}(\boldsymbol{X}), \boldsymbol{X}) = \int_{t_0}^{t} \boldsymbol{P}(\boldsymbol{F}(\boldsymbol{X}), \boldsymbol{X}) : \dot{\boldsymbol{F}} dt \quad \text{and} \quad \dot{\Psi} = \boldsymbol{P} : \dot{\boldsymbol{F}} = \sum_{i,j=1}^{3} \frac{\partial \Psi}{\partial F_{ij}} \dot{F}_{ij}, \tag{A.6}$$

where $\boldsymbol{P}$ is the first Piola-Kirchhoff stress tensor (work conjugate to the rate of deformation gradient $\dot{\boldsymbol{F}}$). Equation (A.6) is often used as a definition for hyperelastic materials [98]. Considering the restrictions imposed by objectivity, Ψ can be expressed in terms of the right

Cauchy-Green deformation tensor $\mathbf{C}$

$$\Psi(\boldsymbol{C}(\boldsymbol{X}),\boldsymbol{X}) = \int_{t_0}^{t} \boldsymbol{S}(\boldsymbol{C}(\boldsymbol{X}),\boldsymbol{X}) : \dot{\boldsymbol{C}} dt \quad \text{and} \tag{A.7}$$

$$\dot{\Psi} = \frac{1}{2}\boldsymbol{S} : \dot{\boldsymbol{F}} = \sum_{i,j=1}^{3} \frac{\partial \Psi}{\partial C_{ij}} \dot{C}_{ij} = \frac{1}{2} \sum_{i,j=1}^{3} \frac{\partial \Psi}{\partial E_{ij}} \dot{C}_{ij}, \tag{A.8}$$

where $\boldsymbol{S}$ is the second Piola-Kirchhoff stress tensor (work conjugate to the rate of right Cauchy-Green deformation $\mathbf{C}$) and $\boldsymbol{E} = 1/2(\boldsymbol{C} - \boldsymbol{I})$ is the Lagrangian strain tensor.

The relationship between $\boldsymbol{S}$ and $\boldsymbol{C}$ or $\boldsymbol{E}$ is nonlinear. A linear relationship can be obtained by taking the directional derivative of $\boldsymbol{S}$

$$\begin{aligned} DS_{IJ}[\boldsymbol{u}] &= \left.\frac{d}{d\epsilon}\right|_{\epsilon=0} S_{IJ}(E_{KL}[\phi + \epsilon \boldsymbol{u}]) \\ &= \sum_{K,L=1}^{3} \frac{\partial S_{IJ}}{\partial E_{KL}} \left.\frac{d}{d\epsilon}\right|_{\epsilon=0} E_{KL}[\phi + \epsilon \boldsymbol{u}] \\ &= \sum_{K,L=1}^{3} \frac{\partial S_{IJ}}{\partial E_{KL}} DE_{KL}[\boldsymbol{u}], \end{aligned} \tag{A.9}$$

where the chain rule has been applied in the second line and $DE_{K,L}[\boldsymbol{u}]$ is the directional derivative of E_{KL}. More concisely, Eq. (A.9) is

$$D\boldsymbol{S}[\boldsymbol{u}] = \mathbb{C} : D\boldsymbol{E}[\boldsymbol{u}], \tag{A.10}$$

where $\mathbb{C}$ is the fourth-order Lagrangian (or material) elasticity tensor

$$\mathbb{C} \equiv \sum_{I,J,K,L=1}^{3} \frac{\partial S_{IJ}}{\partial E_{KL}} \boldsymbol{E}_I \otimes \boldsymbol{E}_J \otimes \boldsymbol{E}_K \otimes \boldsymbol{E}_L = 2\frac{\partial \boldsymbol{S}}{\partial \boldsymbol{C}} = 4\frac{\partial^2 \Psi}{\partial \boldsymbol{C} \partial \boldsymbol{C}}. \tag{A.11}$$

The spatial equivalent of $\mathbb{C}$ can be obtained by applying the push forward operation on the rate form of Eq. (A.10) to obtain [98]

$$\boldsymbol{\sigma}^{o} = \boldsymbol{c} : \boldsymbol{d}, \tag{A.12}$$

where $\boldsymbol{\sigma}^{o}$ is the Truesdell stress rate, $\boldsymbol{d}$ is the rate of deformation tensor, $\boldsymbol{c}$ is the Eulerian

(or spatial) elasticity tensor

$$\boldsymbol{c} \equiv \sum_{i,j,k,l,I,J,K,L=1}^{3} J^{-1} F_{iI} F_{jJ} F_{kK} F_{lL} C_{IJKL} \boldsymbol{e}_i \otimes \boldsymbol{e}_j \otimes \boldsymbol{e}_k \otimes \boldsymbol{e}_l. \tag{A.13}$$

A.3 Isotropic Hyperelasticity

The hyperelastic constitutive equations discussed in the previous section are unrestricted. Here, the constitutive equations are restricted to be isotropic, i.e., the constitutive behavior to be identical in any material direction. Consequently, the elastic potential must be a function of only invariants of $\boldsymbol{C} : \Psi(I_C, II_C, III_C, \boldsymbol{X})$. The invariant of $\boldsymbol{C}$ are

$$\begin{aligned} I_C &= \mathrm{tr}\boldsymbol{C} = \boldsymbol{C} : \boldsymbol{I} \\ II_C &= \mathrm{tr}\boldsymbol{C}\boldsymbol{C} = \boldsymbol{C} : \boldsymbol{C} \\ III_C &= \det\boldsymbol{C} = J^2. \end{aligned} \tag{A.14}$$

As a result of the isotropic restriction, the second Piola-Kirchhoff stress tensor becomes

$$\begin{aligned} \boldsymbol{S} = 2\frac{\partial \Psi}{\partial \boldsymbol{C}} &= 2\frac{\partial \Psi}{\partial I_C}\frac{\partial I_C}{\partial \boldsymbol{C}} + 2\frac{\partial \Psi}{\partial II_C}\frac{\partial II_C}{\partial \boldsymbol{C}} + 2\frac{\partial \Psi}{\partial III_C}\frac{\partial III_C}{\partial \boldsymbol{C}} \\ &= 2\Psi_I \boldsymbol{I} + 4\Psi_{II}\boldsymbol{C} + 2J^2\Psi_{III}\boldsymbol{C}^{-1}, \end{aligned} \tag{A.15}$$

where $\Psi_I = \partial\Psi/\partial I_C, \Psi_{II} = \partial\Psi/\partial II_C$, and $\Psi_{III} = \partial\Psi/\partial III_C$. Applying the relationship $\boldsymbol{\sigma} = J^{-1}\boldsymbol{F}\boldsymbol{S}\boldsymbol{F}^T$, the Cauchy stress can be expressed as

$$\boldsymbol{\sigma} = 2J^{-1}\Psi_I \boldsymbol{b} + 4J^{-1}\Psi_{II}\boldsymbol{b}^2 + 2J\Psi_{III}\boldsymbol{I}. \tag{A.16}$$

A.4 Compressible Neo-Hookean

A compressible Neo-Hookean material is a simple case of an isotropic hyperelastic material. The elastic potential of a compressible Neo-Hookean material is

$$\Psi = \frac{\mu}{2}(I_C - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{A.17}$$

where $J^2 = III_C$ and the constants λ and μ are the Lamé parameters:

$$\begin{aligned} \mu &= \text{Shear Modulus} \\ \lambda &= \frac{2\mu\nu}{1-2\nu} \\ \nu &= \text{Poisson Ratio.} \end{aligned} \tag{A.18}$$

Note that in the absence of deformation (i.e., $\mathbf{C}$=$\mathbf{I}$), $\Psi = 0$. Applying Eq. (A.16) to Eq. (A.17), the Cauchy stress is

$$\begin{aligned} \boldsymbol{\sigma} &= 2J^{-1}\frac{\partial \Psi}{\partial I_C}\boldsymbol{b} + 4J^{-1}\frac{\partial \Psi}{\partial II_C}\boldsymbol{b}^2 + 2J\frac{\partial \Psi}{\partial III_C}\boldsymbol{I} \\ &= 2J^{-1}\left(\frac{\mu}{2}\right)\boldsymbol{b} + 4J^{-1}(0)\boldsymbol{b}^2 + 2J\left(-\frac{\mu}{2}J^{-2} + \frac{\lambda}{2}J^{-2}\ln J\right)\boldsymbol{I} \\ &= \frac{\mu}{J}(\boldsymbol{b}-\boldsymbol{I}) + \frac{\lambda}{J}(\ln J)\boldsymbol{I}. \end{aligned} \tag{A.19}$$

Comparing to the Neo-Hookean constitutive equation used in Lai et al. [99], μ is the shear modulus and $\lambda \equiv (2\mu\nu)/(1-2\nu)$ where ν is Poisson's ratio.

The corresponding Eulerian elasticity tensor for the Neo-Hookean material can obtained by a push forward operation of the Lagrangian elasticity tensor to obtain [98]

$$\boldsymbol{c}_{ijkl} = \lambda'\delta_{ij}\delta_{kl} + \mu'(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}), \tag{A.20}$$

where λ' and μ' are effective Lame moduli

$$\lambda' \equiv \frac{\mu}{J} \quad \text{and} \quad \mu' \equiv \frac{\mu - \lambda\ln J}{J}. \tag{A.21}$$

A.5 Linearized Equilibrium Equations

The internal virtual work can be expressed in a Lagrangian form via

$$\delta W_{int}(\phi, \delta\boldsymbol{v}) = \int_V \boldsymbol{S} : \delta\dot{\boldsymbol{E}} dV, \tag{A.22}$$

demonstrating that the *second Piola-Kirchhoff stress tensor* $\boldsymbol{S}$ is the work conjugate to the *material strain rate tensor* $\dot{\boldsymbol{E}}$. The corresponding directional derivative is

$$\begin{aligned} D\delta W_{int}(\phi, \delta\boldsymbol{v})[\boldsymbol{u}] &= \int_V D(\boldsymbol{S} : \delta\dot{\boldsymbol{E}})[\boldsymbol{u}]dV \\ &= \int_V D\boldsymbol{S}[\boldsymbol{u}] : \delta\dot{\boldsymbol{E}}dV + \int_V \boldsymbol{S} : D\delta\dot{\boldsymbol{E}}[\boldsymbol{u}]dV \\ &= \int_V \delta\dot{\boldsymbol{E}} : \mathbb{C} : D\boldsymbol{E}[\boldsymbol{u}] + \int_V \boldsymbol{S} : D\delta\dot{\boldsymbol{E}}[\boldsymbol{u}]dV \\ &= \int_V \delta\dot{\boldsymbol{E}} : \mathbb{C} : D\boldsymbol{E}[\boldsymbol{u}] + \int_V \boldsymbol{S} : [(\nabla_0\boldsymbol{u})^T\nabla_0\delta\boldsymbol{v}]dV, \end{aligned} \tag{A.23}$$

where the product rule has been used in the second line, the relationship for $D\boldsymbol{S}[\boldsymbol{u}]$ in Eq. (A.10) for the third line, and the relationship

$$D\delta\dot{\boldsymbol{E}}[\boldsymbol{u}] = \frac{1}{2}[(\nabla_0\delta\boldsymbol{v})^T\nabla_0\boldsymbol{u} + (\nabla_0\boldsymbol{u})^T\nabla_0\delta\boldsymbol{v}] = (\nabla_0\boldsymbol{u})^T\nabla_0\delta\boldsymbol{v} \tag{A.24}$$

is used in the last line. Furthermore, noting that $D\boldsymbol{E}[\delta\boldsymbol{v}] = \delta\dot{\boldsymbol{E}}$, Eq. (A.23) can be rewritten as

$$D\delta W_{int}(\phi, \delta\boldsymbol{v})[\boldsymbol{u}] = \int_V D\boldsymbol{E}[\delta\boldsymbol{v}] : \mathbb{C} : D\boldsymbol{E}[\boldsymbol{u}] + \int_V \boldsymbol{S} : [(\nabla_0\boldsymbol{u})^T\nabla_0\delta\boldsymbol{v}]dV. \tag{A.25}$$

The linearized equilibrium equation of Eq. (A.25) can be simplified to expressing in the Eulerian (or spatial) form. This is achieved by push-forward and pull-back operations on the individual terms to obtain (see Section 8.4 of Ref. [98])

$$\begin{aligned} D\boldsymbol{E}[\delta\boldsymbol{v}] : \mathbb{C} : D\boldsymbol{E}[\boldsymbol{u}]dV &= \delta\boldsymbol{d} : \boldsymbol{c} : \varepsilon dv \\ \boldsymbol{S} : [(\nabla_0\boldsymbol{u})^T\nabla_0\delta\boldsymbol{v}]dV &= \boldsymbol{\sigma} : [(\nabla\boldsymbol{u})^T\nabla\delta\boldsymbol{v}], \end{aligned} \tag{A.26}$$

where $\varepsilon = 1/2[\nabla\boldsymbol{u} + (\nabla\boldsymbol{u})^T]$ is the *small-strain tensor*. Applying the relationships in Eq. (A.26), the Eulerian form of the linearized equilibrium equation is

$$D\delta W_{int}(\phi, \delta\boldsymbol{v})[\boldsymbol{u}] = \int_v \delta\boldsymbol{d} : \boldsymbol{c} : \varepsilon dv + \int_v \boldsymbol{\sigma} : [(\nabla\boldsymbol{u})^T\nabla\delta\boldsymbol{v}]dv. \tag{A.27}$$

Note that

$$\delta\boldsymbol{d} = \frac{1}{2}(\nabla\delta\boldsymbol{v} + (\nabla\delta\boldsymbol{v})^T), \tag{A.28}$$

which is the same functional form between ε and $\boldsymbol{u}$. This together with the fact that $\boldsymbol{c}$ and

$\boldsymbol{\sigma}$ are symmetric implies that $\boldsymbol{u}$ and $\delta\boldsymbol{v}$ can be interchanged without altering results.

$$D\delta W_{int}(\phi, \delta\boldsymbol{v})[\boldsymbol{u}] = D\delta W_{int}(\phi, \boldsymbol{u})[\delta\boldsymbol{v}]. \tag{A.29}$$

Such a symmetry gives rise to a symmetric tangent-stiffness matrix upon discretization.

A.6 Discretization of Linearized Equilibrium Equations

A.6.1 Isoparametric Elements

To discretize the linearized virtual work equation in Eq. (A.27), isoparametric elements are used to interpolate the current position in terms of current nodal position $\boldsymbol{x}_a$

$$\boldsymbol{x} = \sum_{a=1}^{n} N_a \boldsymbol{x}_a(t), \tag{A.30}$$

where $N_a(\xi_1, \xi_2, \xi_3)$ are the shape functions and n denotes the number of nodes per element. Differentiating Eq. (A.30) with respect to time gives real and virtual velocity interpolations

$$\boldsymbol{v} = \sum_{a=1}^{n} N_a \boldsymbol{v}_a \quad \text{and} \quad \delta\boldsymbol{v} = \sum_{a=1}^{n} N_a \delta\boldsymbol{v}_a, \tag{A.31}$$

respectively. Similarly, restricting the motion brought about by an arbitrary increment $\boldsymbol{u}$ according Eq. (A.30) implies that the displacement is interpolated as follows

$$\boldsymbol{u} = \sum_{a=1}^{n} N_a \boldsymbol{u}_a. \tag{A.32}$$

Using Eq. (A.31) the real and virtual velocity gradient can also be written in terms of

the shape functions as

$$\begin{aligned}
\boldsymbol{l} = \nabla \boldsymbol{v} &= \sum_{i,j=1}^{3} \frac{\partial v_i}{\partial x_j} \boldsymbol{e}_i \otimes \boldsymbol{e}_j = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \frac{\partial v_1}{\partial x_2} & \frac{\partial v_1}{\partial x_3} \\ \frac{\partial v_2}{\partial x_1} & \frac{\partial v_2}{\partial x_2} & \frac{\partial v_2}{\partial x_3} \\ \frac{\partial v_3}{\partial x_1} & \frac{\partial v_3}{\partial x_2} & \frac{\partial v_3}{\partial x_3} \end{bmatrix} \\
&= \sum_{a=1}^{n} \begin{bmatrix} \frac{\partial N_a}{\partial x_1} v_{1a} & \frac{\partial N_a}{\partial x_2} v_{1a} & \frac{\partial N_a}{\partial x_3} v_{1a} \\ \frac{\partial N_a}{\partial x_1} v_{2a} & \frac{\partial N_a}{\partial x_2} v_{2a} & \frac{\partial N_a}{\partial x_3} v_{2a} \\ \frac{\partial N_a}{\partial x_1} v_{3a} & \frac{\partial N_a}{\partial x_2} v_{3a} & \frac{\partial N_a}{\partial x_3} v_{3a} \end{bmatrix} \\
&= \sum_{a=1}^{n} \boldsymbol{v}_a \otimes \nabla N_a, \qquad \text{(A.33)}
\end{aligned}$$

similarly

$$\delta \boldsymbol{l} = \sum_{a=1}^{n} \delta \boldsymbol{v}_a \otimes N_a. \qquad \text{(A.34)}$$

Furthermore, applying Eq. (A.34) the virtual displacement rate tensor is

$$\delta \boldsymbol{d} = \frac{1}{2} \left(\delta \boldsymbol{l} + \delta \boldsymbol{l}^T \right) = \frac{1}{2} \sum_{a=1}^{n} \left(\delta \boldsymbol{v}_a \otimes \nabla N_a + \nabla N_a \otimes \delta \boldsymbol{v}_a \right). \qquad \text{(A.35)}$$

The deformation gradient tensor, which is used to calculate the Cauchy stress tensor in Eq. (A.19), can also be written in terms of the shape functions as

$$\begin{aligned}
\boldsymbol{F} &= \sum_{i,I=1}^{3} F_{iI} \boldsymbol{e}_i \otimes \boldsymbol{E}_I = \begin{bmatrix} F_{11} & F_{12} & F_{13} \\ F_{21} & F_{22} & F_{23} \\ F_{31} & F_{32} & F_{33} \end{bmatrix}, \quad \text{where} \\
F_{iI} &= \frac{\partial x_i}{\partial X_I} = \sum_{a=1}^{n} x_{a,i} \frac{\partial N_a}{\partial X_{a,I}}. \qquad \text{(A.36)}
\end{aligned}$$

Next consider the the linearized virtual work for element (e) linking nodes a and b. Specifically, the constitutive and initial stress components of Eq. (A.27) are examined separately.

A.6.2 Constitutive Component

The first term on the right-hand-side of Eq. (A.27) is the constitutive component. Its contribution for element (e) linking nodes a and b is

$$D\delta W_c^{(e)}(\phi, N_a\delta\boldsymbol{v}_a)[N_b\boldsymbol{u}_b] = \int_{v^{(e)}} \frac{1}{2}\left(\delta\boldsymbol{v}_a \otimes \nabla N_a + \nabla N_a \otimes \delta\boldsymbol{v}_a\right) : \boldsymbol{c} : \frac{1}{2}\left(\delta\boldsymbol{u}_b \otimes \nabla N_b + \nabla N_b \otimes \delta\boldsymbol{v}_b\right) dv. \tag{A.37}$$

Equation (A.37) can be written in matrix-vector (also known as Voigt) notation by reinterpreting small-strain tensor ε as

$$\hat{\varepsilon} \equiv [\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, 2\varepsilon_{12}, 2\varepsilon_{23}, 2\varepsilon_{13}]^T = \sum_{a=1}^{n} \boldsymbol{B}_a\boldsymbol{u}_a \tag{A.38}$$

and the virtual displacement rate tensor as

$$\delta\hat{\boldsymbol{d}} \equiv [\delta d_{11}, \delta d_{22}, \delta d_{33}, 2\delta d_{12}, 2\delta d_{23}, 2\delta d_{13}]^T = \sum_{a=1}^{n} \boldsymbol{B}_a\delta\boldsymbol{v}_a, \tag{A.39}$$

where $\boldsymbol{B}_a$ is defined in terms of derivatives of the shape functions as

$$\boldsymbol{B}_a \equiv \begin{bmatrix} \frac{\partial N_a}{\partial x_1} & 0 & 0 \\ 0 & \frac{\partial N_a}{\partial x_2} & 0 \\ 0 & 0 & \frac{\partial N_a}{\partial x_3} \\ \frac{\partial N_a}{\partial x_2} & \frac{\partial N_a}{\partial x_1} & 0 \\ 0 & \frac{\partial N_a}{\partial x_3} & \frac{\partial N_a}{\partial x_2} \\ \frac{\partial N_a}{\partial x_3} & 0 & \frac{\partial N_a}{\partial x_1} \end{bmatrix}. \tag{A.40}$$

Note that the definition of $\boldsymbol{B}_a$ is different from Ref. [98] because the ordering of the small-strain and virtual displacement rate tensor components in Eqs. (A.38) and (A.39) is different

from what is presented in Ref. [98]. Applying Voigt notation, Eq. (A.37) is simplified to

$$\begin{aligned} D\delta W_c^{(e)}(\phi, N_a\delta\boldsymbol{v}_a)[N_b\boldsymbol{u}_b] &= \int_{v^{(e)}} (\boldsymbol{B}_a\delta\boldsymbol{v}_a)^T\boldsymbol{D}(\boldsymbol{B}_b\boldsymbol{u}_b)dv \\ &= \delta\boldsymbol{v}_a\cdot\left(\int_{v^{(e)}}\boldsymbol{B}_a^T\boldsymbol{D}\boldsymbol{B}_b dv\right)\boldsymbol{u}_b \\ &= \delta\boldsymbol{v}_a\cdot\boldsymbol{K}_{c,ab}^{(e)}\boldsymbol{u}_a, \end{aligned} \tag{A.41}$$

where

$$\boldsymbol{D} = \begin{bmatrix} \lambda'+2\mu' & \lambda' & \lambda' & 0 & 0 & 0 \\ \lambda' & \lambda'+2\mu' & \lambda' & 0 & 0 & 0 \\ \lambda' & \lambda' & \lambda'+2\mu' & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu' & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu' & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu' \end{bmatrix}, \quad \text{where} \quad \lambda' = \frac{\lambda}{\det\boldsymbol{J}}\ ,\ \mu' = \frac{\mu-\lambda\ln(\det\boldsymbol{J})}{\det\boldsymbol{J}}, \tag{A.42}$$

for a Neo-Hookean material described by Eq. (A.17) and $\boldsymbol{K}_{c,ab}^{(e)}$ is the constitutive component of the tangent stiffness matrix relating nodes a and b in element (e).

A.6.3 Initial Stress (or Geometric) Component

The second term on the right-hand-side of Eq. (A.27) is the initial stress component. Its contribution for element (e) linking nodes a and b is

$$\begin{aligned} D\delta W_\sigma^{(e)}(\phi, N_a\delta\boldsymbol{v}_a)[N_b\boldsymbol{u}_b] &= \int_{v^{(e)}}\boldsymbol{\sigma} : [(\nabla\boldsymbol{u}_b)^T\nabla\delta\boldsymbol{v}_a]dv \\ &= \int_{v^{(e)}}\boldsymbol{\sigma} : [(\boldsymbol{u}_b\otimes\nabla N_b)^T(\delta\boldsymbol{v}_a\otimes\nabla N_a)]dv \\ &= \int_{v^{(e)}}\boldsymbol{\sigma} : [(\delta\boldsymbol{v}_a\cdot\boldsymbol{u}_b)\nabla N_b\otimes\nabla N_a]dv \\ &= (\delta\boldsymbol{v}_a\cdot\boldsymbol{u}_b)\int_{v^{(e)}}\nabla N_a\cdot\boldsymbol{\sigma}\nabla N_b dv, \end{aligned} \tag{A.43}$$

where the relationship $\boldsymbol{\sigma} : (\boldsymbol{u}\otimes\boldsymbol{v}) = \boldsymbol{u}\cdot\boldsymbol{\sigma}\boldsymbol{v}$ has been used in the last line. Note that $\nabla\boldsymbol{u}$ and $\nabla\delta\boldsymbol{v}$ are tensors (see definition in Eq. (A.33)).

Observing that $\int_{v^{(e)}}\nabla N_a\cdot\boldsymbol{\sigma}\nabla N_b dv$ is a scalar and that $\delta\boldsymbol{v}_a\cdot\boldsymbol{u}_b = \delta\boldsymbol{v}_a\cdot\boldsymbol{I}\boldsymbol{u}_b$, Eq. (A.43)

can be written in matrix form as

$$\begin{aligned} D\delta W_{\sigma}^{(e)}(\phi, N_a\delta\boldsymbol{v}_a)[N_b\boldsymbol{u}_b] &= \delta\boldsymbol{v}_a \cdot \left(\int_{v^{(e)}} \nabla N_a \cdot \boldsymbol{\sigma}\nabla N_b dv \boldsymbol{I}\right)\boldsymbol{u}_b \\ &= \delta\boldsymbol{v}_a \cdot \boldsymbol{K}_{\sigma,ab}^{(e)}\boldsymbol{u}_b, \end{aligned} \tag{A.44}$$

where $\boldsymbol{K}_{\sigma,ab}^{(e)}$ is the initial-stress component of the tangent stiffness matrix relating nodes a and b in element (e).

A.7 Derivation of Residual Form for NonLinear FEM

To apply the Newton-Raphson iteration procedure to Nonlinear FEM, one needs to calculate the elasticity tensor. As shown above, the elasticity tensor is a 4-tensor that is determined by taking directional derivatives of a potential energy with respect to second-order tensors. Often times, it is non-trivial to handle the high-dimensionality of the elasticity tensor. To bypass the need to handle the elasticity tensor, one can use auto differentiation, which is implemented based on the residual form of the discretized virtual work. Since the residual form of the discretized virtual work involves only up to a 2-tensor, the Cauchy stress-tensor. Therefore, in this section the residual form of the discretized virtual work is derived in detail.

A.7.1 Principle of Virtual Work

Based on chapter 5 of Ref. [98], the spatial virtual work equation is

$$\delta W(\phi, \delta\boldsymbol{v}) = \int_v \boldsymbol{\sigma} : \delta\boldsymbol{d}dv - \int_v \boldsymbol{f}\cdot\delta\boldsymbol{v}dv - \int_{\partial v} \boldsymbol{t}\cdot\delta\boldsymbol{v}da = 0, \tag{A.45}$$

where $\boldsymbol{\sigma}$ is the Cauchy stress-tensor (2-tensor), $\boldsymbol{f}$ is the body force per unit volume, and $\boldsymbol{t}$ is the traction force per unit area (both 1-tensors). Note that Eq. (A.45) is a functional of a trial solution ϕ (a trial configuration) and virtual velocity $\delta\boldsymbol{v}$. In order to use FEM, Eq. (A.45) must be discretized. Such a discretization can be obtained by interpolating the current position with isoparametric elements (see Eq. (A.30)).

A.7.2 Discretization

Analogous to chapter 9 of [98], the discretization of Eq. (A.45) is obtained by considering the contribution to $\delta W(\phi, \delta \boldsymbol{v})$ caused by a single virtual nodal velocity of element (e) $\delta \boldsymbol{v}_a^{(e)}$

$$\delta W_a^{(e)}(\phi, N_a \delta \boldsymbol{v}_a^{(e)}) = \int_{v^{(e)}} \boldsymbol{\sigma} : (\delta \boldsymbol{v}_a^{(e)} \otimes \nabla N_a) dv - \int_{v^{(e)}} \boldsymbol{f} \cdot N_a \delta \boldsymbol{v}_a^{(e)} dv - \int_{\partial v^{(e)}} \boldsymbol{t} \cdot N_a \delta \boldsymbol{v}_a^{(e)} da = 0, \tag{A.46}$$

where the interpolations of Eqs. (A.31) and (A.35) have been used for $\delta \boldsymbol{v}$ and $\delta \boldsymbol{d}$, respectively, in Eq. (A.45). Note that the symmetry of $\boldsymbol{\sigma}$ has been used to simplify the interpolation of Eq. (A.35).

Observing that $\delta \boldsymbol{v}$ is independent of the integration and using the tensor relationship $\boldsymbol{\sigma} : (\delta \boldsymbol{v}_a \otimes \nabla N_a) = \delta \boldsymbol{v}_a \cdot \boldsymbol{\sigma} \nabla N_a$ (see Eq. (2.52b) of Ref. [98]), Eq. (A.46) can be simplified to

$$\begin{aligned} \delta W_a^{(e)}(\phi, N_a \delta \boldsymbol{v}_a^{(e)}) &= \delta \boldsymbol{v}_a^{(e)} \cdot \left(\int_{v^{(e)}} \boldsymbol{\sigma} \nabla N_a dv - \int_{v^{(e)}} \boldsymbol{f} \cdot N_a dv - \int_{\partial v^{(e)}} \boldsymbol{t} \cdot N_a da \right) \\ &= \delta \boldsymbol{v}_a^{(e)} \cdot \left(\boldsymbol{T}_a^{(e)} - \boldsymbol{F}_a^{(e)} \right) = 0, \end{aligned} \tag{A.47}$$

where

$$\boldsymbol{T}_a^{(e)} \equiv \int_{v^{(e)}} \boldsymbol{\sigma} \nabla N_a dv \quad \text{and } \boldsymbol{F}_a^{(e)} \equiv \int_{v^{(e)}} \boldsymbol{f} \cdot N_a dv + \int_{\partial v^{(e)}} \boldsymbol{t} \cdot N_a da. \tag{A.48}$$

The contribution to $\delta W(\phi, \delta \boldsymbol{v})$ from all nodes belonging to an element (e) is then

$$\begin{aligned} \delta W^{(e)}(\phi, \sum_{a=1}^{n} N_a \delta \boldsymbol{v}_a^{(e)}) &= \sum_{a=1}^{n} \delta \boldsymbol{v}_a^{(e)} \cdot \left(\boldsymbol{T}_a^{(e)} - \boldsymbol{F}_a^{(e)} \right) \\ &= \delta \boldsymbol{v}^{(e)} \cdot \left(\boldsymbol{T}^{(e)} - \boldsymbol{F}^{(e)} \right) = 0, \end{aligned} \tag{A.49}$$

where n is the number of nodes in element (e). Since Eq. (A.49) must hold for arbitrary values of $\delta \boldsymbol{v}^{(e)}$ in each element of the FEM mesh, the elemental residual force emerges as

$$\boldsymbol{R}^{(e)} = \boldsymbol{T}^{(e)} - \boldsymbol{F}^{(e)}. \tag{A.50}$$

Note that the integrators in the biotissue code operate on the element level where $\boldsymbol{T}^{(e)}$ and $\boldsymbol{F}^{(e)}$ are calculated.

A.7.3 Implementation Details of the Internal Elemental Forces, $\boldsymbol{T}^{(e)}$

From Eqs. (A.48) and (A.49), the internal elemental forces are

$$\begin{aligned} \boldsymbol{T}^{(e)} &= \sum_{a=1}^{n} \boldsymbol{T}_a^{(e)} = \sum_{a=1}^{n} \int_{v^{(e)}} \boldsymbol{\sigma} \nabla N_a dv \\ &= \boldsymbol{\sigma} \sum_{a=1}^{n} \sum_{i=1}^{n_g} W_i \nabla N_a(r_i, s_i, t_i) \det\left(\boldsymbol{J}^{(e)}(r_i, s_i, t_i)\right), \end{aligned} \tag{A.51}$$

where the volume integral has been replaced by a direct (non-tensor-product) Gauss quadrature formula. The variables n_g is the number of Gauss quadrature points, W_i is the weight at the i^{th} quadrature point, (r_i, s_i, t_i) are the coordinates of the parent domain at the i^{th} quadrature point, and $\boldsymbol{J}^{(e)}$ is the Jacobian of element (e).

Specific to the biotissue multi-scale code, an integrator acts on the each Gauss quadrature point within an element. This integrator is responsible for calculating the term

$$\begin{aligned} \mathbb{I}_i &\equiv \boldsymbol{\sigma} \sum_{a=1}^{n} \nabla N_a(r_i, s_i, t_i) \det\left(\boldsymbol{J}^{(e)}(r_i, s_i, t_i)\right) \\ &= \det\left(\boldsymbol{J}^{(e)}(r_i, s_i, t_i)\right) \sum_{a=1}^{n} \begin{bmatrix} \sigma_{11}\frac{\partial N_a}{\partial x} + \sigma_{12}\frac{\partial N_a}{\partial y} + \sigma_{13}\frac{\partial N_a}{\partial z} \\ \sigma_{21}\frac{\partial N_a}{\partial x} + \sigma_{22}\frac{\partial N_a}{\partial y} + \sigma_{23}\frac{\partial N_a}{\partial z} \\ \sigma_{31}\frac{\partial N_a}{\partial x} + \sigma_{32}\frac{\partial N_a}{\partial y} + \sigma_{33}\frac{\partial N_a}{\partial z} \end{bmatrix} \end{aligned} \tag{A.52}$$

and internal elemental forces becomes

$$\boldsymbol{T}^{(e)} = \sum_{i=1}^{n_g} \mathbb{I}_i, \tag{A.53}$$

The external elemental forces from Eq. (A.50) are implemented directly onto the degrees of freedom of the global forcing vector during the linear system assembly stage.

A.8 Governing Equation

The divergence of the macroscopic stress tensor, $\boldsymbol{\sigma}$ is derived by applying the Reynolds transport theorem to the volume-averaged stress and invoking microscopic equilibrium:

$$\sigma_{ij,i}^{(M)} = \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} \left(s_{ij}^{(m)} - \sigma_{ij}^{(M)}\right) u_{k,i}^{(m)} n_k dA^{(m)}, \tag{A.54}$$

where the superscript m is applied to microscale quantities, and M is applied to macroscale quantities. Also, $u_k^{(m)}$ is the displacement of the RVE boundary on the microscale, n_k is the unit normal vector, and $s_{ij}^{(m)}$ are elements of the microscopic stress tensor [48], [100]. A volume averaging of $s_{ij}^{(m)}$ is employed to obtain $\sigma_{ij}^{(M)}$

$$\sigma_{ij}^{(M)} = \frac{1}{V^{(m)}} \int_{V^{(m)}} s_{ij}^{(m)} dV^{(m)}, \tag{A.55}$$

where $V^{(m)}$ is the microscale volume [100].

From Eq. (A.54) we see that the conservation of momentum equation becomes:

$$\sigma_{ij,i}^{(M)} - \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} \left(s_{ij}^{(m)} - \sigma_{ij}^{(M)} \right) u_{k,i}^{(m)} n_k dA^{(m)} = 0, \tag{A.56}$$

The virtual work, δW, is obtained by multiplying Eq. (A.56) by a virtual velocity, $\delta \boldsymbol{v}$ (also known as a test function), and integrating over the current volume:

$$\begin{aligned}
\delta W &= \int_{V^{(M)}} \left(\sigma_{ij,i}^{(M)} - \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} \left(s_{ij}^{(m)} - \sigma_{ij}^{(M)} \right) u_{k,i}^{(m)} n_k dA^{(m)} \right) \delta v_j^{(M)} dV^{(M)} \\
&= \int_{V_0^{(M)}} \left(\sigma_{ij,i}^{(M)} - \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} \left(s_{ij}^{(m)} - \sigma_{ij}^{(M)} \right) u_{k,i}^{(m)} n_k dA^{(m)} \right) \delta v_j^{(M)} J^{(M)} dV_0^{(M)} \\
&= \int_{V_0^{(M)}} \left(\sigma_{ij,i}^{(M)} - Q_j^{(m/M)} \right) \delta v_j^{(M)} J^{(M)} dV_0^{(M)} \\
&= \int_{V_0^{(M)}} \left(\mathrm{div} \boldsymbol{\sigma}^{(M)} - \boldsymbol{Q}^{(m/M)} \right) \cdot \delta \boldsymbol{v}^{(M)} J^{(M)} dV_0^{(M)},
\end{aligned} \tag{A.57}$$

where $J^{(M)}$ is the Jacobian of the macroscale, and:

$$Q_j^{(m/M)} = \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} \left(s_{ij}^{(m)} - \sigma_{ij}^{(M)} \right) u_{k,i}^{(m)} n_k dA^{(m)}. \tag{A.58}$$

Note that the superscript (m/M) indicates the quantity involves variables from both the micro and macro scales.

Using the identity

$$\mathrm{div}(\boldsymbol{\sigma} \cdot \delta \boldsymbol{v} J) = (\boldsymbol{\sigma} : \nabla \delta \boldsymbol{v}) J + (\delta \boldsymbol{v} \cdot \mathrm{div} \boldsymbol{\sigma}) J + (\boldsymbol{\sigma} \cdot \delta \boldsymbol{v}) \cdot \nabla J, \tag{A.59}$$

Eq. (A.57) becomes

$$\begin{aligned}\delta W &= \int_{V_0^{(M)}} \left(\mathrm{div}(\boldsymbol{\sigma}^{(M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)}) - (\boldsymbol{\sigma}^{(M)} : \nabla\delta\boldsymbol{v}) J^{(M)} - (\boldsymbol{\sigma}^{(M)} \cdot \delta\boldsymbol{v}^{(M)}) \cdot \nabla J^{(M)}\right) dV_0^{(M)} \\ &- \int_{V_0^{(M)}} \boldsymbol{Q}^{(m/M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)} dV_0^{(M)} \end{aligned} \tag{A.60}$$

Further applying the divergence theorem we have

$$\begin{aligned}\delta W &= \int_{\partial V_0^{(M)}} \boldsymbol{n} \cdot \boldsymbol{\sigma}^{(M)} \delta\boldsymbol{v}^{(M)} J^{(M)} dA_0^{(M)} \\ &- \int_{V_0^{(M)}} \left((\boldsymbol{\sigma}^{(M)} : \nabla\delta\boldsymbol{v}) J^{(M)} + (\boldsymbol{\sigma}^{(M)} \cdot \delta\boldsymbol{v}^{(M)}) \cdot \nabla J^{(M)}\right) dV_0^{(M)} \\ &- \int_{V_0^{(M)}} \boldsymbol{Q}^{(m/M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)} dV_0^{(M)} \\ &= \int_{\partial V_0^{(M)}} \boldsymbol{t}^{(M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)} dA_0^{(M)} \\ &- \int_{V_0^{(M)}} \left((\boldsymbol{\sigma}^{(M)} : \nabla\delta\boldsymbol{v}) J^{(M)} + (\boldsymbol{\sigma}^{(M)} \cdot \delta\boldsymbol{v}^{(M)}) \cdot \nabla J^{(M)}\right) dV_0^{(M)} \\ &- \int_{V_0^{(M)}} \boldsymbol{Q}^{(m/M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)} dV_0^{(M)} \\ &= \int_{V_0^{(M)}} \left((\boldsymbol{\sigma}^{(M)} : \delta\boldsymbol{d}) J^{(M)} + (\boldsymbol{\sigma}^{(M)} \cdot \delta\boldsymbol{v}^{(M)}) \cdot \nabla J^{(M)} + \boldsymbol{Q}^{(m/M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)}\right) dV_0^{(M)} \\ &- \int_{\partial V_0^{(M)}} \boldsymbol{t}^{(M)} \cdot \delta\boldsymbol{v}^{(M)} J^{(M)} dA_0^{(M)} \\ &= \delta W_{int} - \delta W_{ext} = 0 \end{aligned} \tag{A.61}$$

where the virtual rate of deformation tensor, $\delta\boldsymbol{d}^{(M)}$, can be used instead of $\nabla\delta\boldsymbol{v}$ in the double contraction with the Cauchy stress due to the symmetry of the Cauchy stress σ. Also, $\boldsymbol{t}^{(M)}$ is the traction vector.

A.9 Newton-Raphson Method and the Tangent-Stiffness Matrix

The Newton-Raphson procedure for solving a nonlinear set of equations is [98]

$$\boldsymbol{K}(\boldsymbol{u}^k)\delta\boldsymbol{u} = -\boldsymbol{R}(\boldsymbol{u}^k),\ \boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta\boldsymbol{u}, \tag{A.62}$$

where $\boldsymbol{R}$ is the out-of-balance force, $\boldsymbol{K}$ is the tangent stiffness matrix, and the superscript k indicates the pseudo time step of the procedure. The left-hand-side of Eq. (A.62) is determined from the directional derivative of $\boldsymbol{R}$ with respect to $\boldsymbol{u}^k$ and in the direction of $\delta\boldsymbol{u}$:

$$\begin{aligned}
\boldsymbol{K}(\boldsymbol{u}^k)\delta\boldsymbol{u} &= D\boldsymbol{R}(\boldsymbol{u}^k)[\delta\boldsymbol{u}] \\
&= \left.\frac{d}{d\epsilon}\right|_{\epsilon=0} \boldsymbol{R}(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u}) \\
&= \left.\frac{d\boldsymbol{R}}{d\boldsymbol{u}^k}\frac{d(\boldsymbol{u}^k + \epsilon\delta\boldsymbol{u})}{d\epsilon}\right|_{\epsilon=0} \\
&= \frac{d\boldsymbol{R}}{d\boldsymbol{u}^k}\delta\boldsymbol{u}. \qquad \text{(A.63)}
\end{aligned}$$

In matrix form we have

$$[\boldsymbol{K}(\boldsymbol{u}^k)][\delta\boldsymbol{u}] = \begin{bmatrix} \frac{\partial R_1}{\partial u_1^k} & \frac{\partial R_1}{\partial u_2^k} & \frac{\partial R_1}{\partial u_3^k} \\ \frac{\partial R_2}{\partial u_1^k} & \frac{\partial R_2}{\partial u_2^k} & \frac{\partial R_2}{\partial u_3^k} \\ \frac{\partial R_3}{\partial u_1^k} & \frac{\partial R_3}{\partial u_2^k} & \frac{\partial R_3}{\partial u_3^k} \end{bmatrix} \begin{bmatrix} \delta u_1 \\ \delta u_2 \\ \delta u_3 \end{bmatrix}, \quad \text{where} \quad [\boldsymbol{R}] = \begin{bmatrix} R_1 \\ R_2 \\ R_3 \end{bmatrix}. \qquad \text{(A.64)}$$

To apply the Newton-Raphson procedure to our problem, the virtual work in Eq. (A.57) must be linearized by taking the directional derivative of δW with respect to $\delta\boldsymbol{v}$ in the direction of $\delta\boldsymbol{u}$ at the current coordinates $\boldsymbol{x}$

$$D\delta W(\boldsymbol{x}, \delta\boldsymbol{v})[\delta\boldsymbol{u}] = D\delta W_{int}(\boldsymbol{x}, \delta\boldsymbol{v})[\delta\boldsymbol{u}] - D\delta W_{ext}(\boldsymbol{x}, \delta\boldsymbol{v})[\delta\boldsymbol{u}]. \qquad \text{(A.65)}$$

Note that $\boldsymbol{t}$ and $\boldsymbol{Q}$ in Eq. (A.61) are assumed to be constant under a virtual displacement $\delta\boldsymbol{u}$.

The linearized virtual work in the biotissue code is

$$\begin{aligned}
D\delta W(\boldsymbol{x}, \delta\boldsymbol{v}^{(M)})[\delta\boldsymbol{u}^{(M)}] &= \int_{V^{(M)}} \left(\delta d_{ij}^{(M)} c_{ijkl}^{(M)} \varepsilon_{kl}^{(M)} + \sigma_{ij}^{(M)} \frac{\partial \delta u_j^{(M)}}{\partial x_i^{(M)}} \frac{\partial \delta v_i^{(M)}}{\partial x_j^{(M)}} \right) dV \\
&- \int_{\partial V^{(M)}} t_i^{(M)} \delta v_i^{(M)} dA^{(M)} - Q_i^{(M)} \delta v_j^{(M)}, \qquad \text{(A.66)}
\end{aligned}$$

where $\boldsymbol{e}$ and $\boldsymbol{\varepsilon}$ are the Eulerian (or Almansi) strain tensor and small-strain tensor, respectively, and $\delta\boldsymbol{d}$ is the virtual rate of deformation tensor:

$$\boldsymbol{e} = \frac{1}{2}(\boldsymbol{I} - \boldsymbol{b}^{-1})$$
$$\boldsymbol{\varepsilon} = \frac{1}{2}\left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\right)$$
$$\boldsymbol{b} = \boldsymbol{F}\boldsymbol{F}^T \tag{A.67}$$
$$\delta \boldsymbol{d} = \frac{1}{2}\left(\nabla \delta \boldsymbol{v} + (\nabla \delta \boldsymbol{v})^T\right). \tag{A.68}$$

The linearization in Eq. (A.66) can be cast into the framework of the Newton-Raphson procedure in Eq. (A.62) by discretization via shape functions:

$$\begin{aligned}
\delta \boldsymbol{u}^{(M)} &= \sum_{a=1}^{n} N_a \boldsymbol{u}_a^{(M)} \\
\delta \boldsymbol{v}^{(M)} &= \sum_{a=1}^{n} N_a \delta \boldsymbol{v}_a^{(M)} \\
\delta \boldsymbol{d}^{(M)} &= \frac{1}{2}\sum_{a=1}^{n} (\delta \boldsymbol{v}_a^{(M)} \otimes \nabla N_a + \nabla N_a \otimes \delta \boldsymbol{v}_a^{(M)}) \\
\delta \boldsymbol{\varepsilon}^{(M)} &= \frac{1}{2}\sum_{a=1}^{n} (\boldsymbol{u}_a^{(M)} \otimes \nabla N_a + \nabla N_a \otimes \boldsymbol{u}_a^{(M)}),
\end{aligned} \tag{A.69}$$

where N_a is the a^{th} node of the shape function and n is the number of nodes contained in each shape function. The discretized form of Eq. (A.66) is obtained by applying the approximations in Eq. (A.69)

$$\begin{aligned}
D\delta W(\boldsymbol{x}, N_a \delta \boldsymbol{v}_a^{(M)})[N_b \delta \boldsymbol{u}_b^{(M)}] \;=\; & \delta v_i^{(M),a} \left(\int_{V^{(e)}} \frac{\partial N_a}{\partial x_j^{(M)}} \right. \\
& \left. c_{ijkl}^{(M)} \frac{\partial N_b}{\partial x_l^{(M)}} dV^{(e)} \right) \delta u_k^{(M),b}
\end{aligned} \tag{A.70}$$

In the multiscale formulation, the Cauchy stresses and their derivatives in Eq. (A.66):

$$\sigma_{ij} \text{ and } \frac{\partial \sigma_{ij}}{\partial e_{kl}}, \tag{A.71}$$

are evaluated from calculations at the microscale representative volume elements (RVEs).

In order to write the stress derivative in matrix form, Voigt notation is used.

$$[\sigma_{(i)}] \equiv \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix}, \quad [e_{(i)}] \equiv \begin{bmatrix} e_{11} \\ e_{22} \\ e_{33} \\ e_{23} \\ e_{13} \\ e_{12} \end{bmatrix}, \text{ and } \left[\frac{\partial \sigma_{(i)}}{\partial e_{(j)}}\right] \equiv \begin{bmatrix} \frac{\partial \sigma_{11}}{\partial e_{11}} & \frac{\partial \sigma_{11}}{\partial e_{22}} & \frac{\partial \sigma_{11}}{\partial e_{33}} & \frac{\partial \sigma_{11}}{\partial e_{23}} & \frac{\partial \sigma_{11}}{\partial e_{13}} & \frac{\partial \sigma_{11}}{\partial e_{12}} \\ \frac{\partial \sigma_{22}}{\partial e_{11}} & \frac{\partial \sigma_{22}}{\partial e_{22}} & \frac{\partial \sigma_{22}}{\partial e_{33}} & \frac{\partial \sigma_{22}}{\partial e_{23}} & \frac{\partial \sigma_{22}}{\partial e_{13}} & \frac{\partial \sigma_{22}}{\partial e_{12}} \\ \frac{\partial \sigma_{33}}{\partial e_{11}} & \frac{\partial \sigma_{33}}{\partial e_{22}} & \frac{\partial \sigma_{33}}{\partial e_{33}} & \frac{\partial \sigma_{33}}{\partial e_{23}} & \frac{\partial \sigma_{33}}{\partial e_{13}} & \frac{\partial \sigma_{33}}{\partial e_{12}} \\ \frac{\partial \sigma_{23}}{\partial e_{11}} & \frac{\partial \sigma_{23}}{\partial e_{22}} & \frac{\partial \sigma_{23}}{\partial e_{33}} & \frac{\partial \sigma_{23}}{\partial e_{23}} & \frac{\partial \sigma_{23}}{\partial e_{13}} & \frac{\partial \sigma_{23}}{\partial e_{12}} \\ \frac{\partial \sigma_{13}}{\partial e_{11}} & \frac{\partial \sigma_{13}}{\partial e_{22}} & \frac{\partial \sigma_{13}}{\partial e_{33}} & \frac{\partial \sigma_{13}}{\partial e_{23}} & \frac{\partial \sigma_{13}}{\partial e_{13}} & \frac{\partial \sigma_{13}}{\partial e_{12}} \\ \frac{\partial \sigma_{12}}{\partial e_{11}} & \frac{\partial \sigma_{12}}{\partial e_{22}} & \frac{\partial \sigma_{12}}{\partial e_{33}} & \frac{\partial \sigma_{12}}{\partial e_{23}} & \frac{\partial \sigma_{12}}{\partial e_{13}} & \frac{\partial \sigma_{12}}{\partial e_{12}} \end{bmatrix} \quad \text{(A.72)}$$

where the parenthesis around the subscripts indicate the use of Voigt notation.

APPENDIX B
Soft Tissue Micro-scale Derivation

B.1 Microscale Forces on Fiber Nodes

Prior to the volume averaging described in Eq. (C.1), the forces that act on the fiber nodes in the RVE must be calculated and are determined from the equilibrium nodal positions of each fiber in the RVE via

$$F \equiv \frac{EA}{B}[\exp(B\varepsilon) - 1], \tag{B.1}$$

where E is the linear modulus of the fiber, A is the cross-sectional area of the fiber, B is a constant, and ε is the fiber's Green strain (along its axis).

Rotating the green strain along the principal axis of the truss, we see that the green strain can be written as:

$$\varepsilon \equiv \frac{1}{2}\left(\lambda^2 - 1\right), \tag{B.2}$$

where λ is the fiber stretch ratio [100].

The force on fiber node b, $\boldsymbol{T}^b$, is defined in terms of the scalar F as

$$\boldsymbol{T}^b = F\boldsymbol{n}, \tag{B.3}$$

where $\boldsymbol{T}^b = T_i^b \boldsymbol{e}_i$ and the unit vector $\boldsymbol{n}$ is determined from the current end coordinates, $\boldsymbol{x}^a$ and $\boldsymbol{x}^b$ at nodes a and b of a fiber, respectively, by

$$\boldsymbol{n} = \frac{1}{l(\boldsymbol{x})}(\boldsymbol{x}^b - \boldsymbol{x}^a), \tag{B.4}$$

where $\boldsymbol{x} = x_i \boldsymbol{e}_i$, the superscript indicates the node of which the position describes, and $l(\boldsymbol{x})$ is the current length of the fiber given by

$$l(\boldsymbol{x}) = \sqrt{(\boldsymbol{x}^b - \boldsymbol{x}^a) \cdot (\boldsymbol{x}^b - \boldsymbol{x}^a)}. \tag{B.5}$$

This chapter authored in conjunction with: J. Merson, V. W. L. Chan

Note that the definitions in Eqs. (B.3) through (B.5) are based on the convention that the force acting on nodes b and a are positive and negative, respectively, when the corresponding fiber element is loaded in tension.

Consequently, $\boldsymbol{T}^a = -\boldsymbol{T}^b$. The equilibrium fiber node positions in a RVE ($\boldsymbol{x}^b$ and $\boldsymbol{x}^a$ of each fiber) are determined via a Newton-Raphson method.

B.2 Microscale Tangent Stiffness Tensor

In order to find the equilibrium fiber-node positions in the RVEs via the Newton-Raphson procedure, we need to define the tangent stiffness tensor. The tangent stiffness tensor is determined by taking the directional derivative of $\boldsymbol{T}^b$ (we can equivalently take the directional derivative of $\boldsymbol{T}^a$, in which case the result will be equal and opposite) in the direction of the displacement of the fiber, $\boldsymbol{u} \equiv \boldsymbol{u}^b - \boldsymbol{u}^a$:

$$\begin{aligned} D\boldsymbol{T}^b[\boldsymbol{u}] &= DF\boldsymbol{n}[\boldsymbol{u}] \\ &= \frac{EA}{B}\Big(D\{\exp(B\varepsilon)-1\}[\boldsymbol{u}]\boldsymbol{n} + D\boldsymbol{n}[\boldsymbol{u}]\{\exp(B\varepsilon)-1\}\Big) \\ &= \frac{EA}{B}\Big(R\boldsymbol{n} + \boldsymbol{S}\{\exp(B\boldsymbol{\varepsilon})-1\}\Big), \end{aligned} \tag{B.6}$$

where the notation $Df(\boldsymbol{x})[\boldsymbol{u}]$ denotes the directional derivative of $f(\boldsymbol{x})$ in the direction of $\boldsymbol{u}$, and

$$R \equiv D\{\exp(B\varepsilon)-1\}[\boldsymbol{u}] \;\text{ and }\; \boldsymbol{S} \equiv D\boldsymbol{n}[\boldsymbol{u}]. \tag{B.7}$$

To obtain a more useful form of Eq. (B.6), $\boldsymbol{R}$ and $\boldsymbol{S}$ can be written out explicitly by expanding the directional derivative and using $\lambda \equiv l(\boldsymbol{x})/L$, and Eq. (B.2):

$$\begin{aligned} D\{\exp(B\varepsilon)-1\}[\boldsymbol{u}] &= B\exp(B\varepsilon)D\varepsilon[\boldsymbol{u}] \\ &= \frac{0.5B}{L^2}\exp(B\varepsilon)D(l^2(\boldsymbol{x})-1)[\boldsymbol{u}] \\ &= \frac{B}{L^2}\exp(B\varepsilon)l(\boldsymbol{x})\boldsymbol{n}\cdot(\boldsymbol{u}^b-\boldsymbol{u}^a) \\ &= R \end{aligned} \tag{B.8}$$

We also have:

$$\begin{aligned} D\boldsymbol{n}[\boldsymbol{u}] &= D\frac{1}{l(\boldsymbol{x})}(\boldsymbol{x}^b - \boldsymbol{x}^a)[\boldsymbol{u}] \\ &= D\frac{1}{l(\boldsymbol{x})}[\boldsymbol{u}](\boldsymbol{x}^b - \boldsymbol{x}^a) + D(\boldsymbol{x}^b - \boldsymbol{x}^a)[\boldsymbol{u}]\frac{1}{l(\boldsymbol{x})} \\ &= -\frac{1}{l^2(\boldsymbol{x})}\boldsymbol{n}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a)(\boldsymbol{x}^b - \boldsymbol{x}^a) + (\boldsymbol{u}^b - \boldsymbol{u}^a)\frac{1}{l(\boldsymbol{x})} \\ &= \frac{(\boldsymbol{u}^b - \boldsymbol{u}^a) - \boldsymbol{n}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a)\boldsymbol{n}}{l(\boldsymbol{x})} \\ &= \boldsymbol{S}. \end{aligned} \tag{B.9}$$

Finally, substituting Eqs. (B.8) and (B.9) into Eq. (B.6) we arrive at:

$$\begin{aligned} D\boldsymbol{T}^b[\boldsymbol{u}] &= \frac{EA}{B}\Bigg[\left(\frac{B}{L^2}\exp(B\varepsilon)l(\boldsymbol{x})\boldsymbol{n}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a)\right)\boldsymbol{n} \\ &+ \left(\frac{(\boldsymbol{u}^b - \boldsymbol{u}^a) - \boldsymbol{n}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a)\boldsymbol{n}}{l(\boldsymbol{x})}\right)\{\exp(B\varepsilon) - 1\}\Bigg] \\ &= \left[\frac{EA}{L^2}\exp(B\varepsilon)l(\boldsymbol{x}) - \frac{F}{l(\boldsymbol{x})}\right](\boldsymbol{n}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a)\boldsymbol{n}) + \frac{F}{l(\boldsymbol{x})}(\boldsymbol{u}^b - \boldsymbol{u}^a) \\ &= \left[\frac{EAl(\boldsymbol{x})}{L^2}\exp(B\varepsilon) - \frac{F}{l(\boldsymbol{x})}\right](\boldsymbol{n}\otimes\boldsymbol{n})_{3\times3}(\boldsymbol{u}^b - \boldsymbol{u}^a) + \frac{F}{l(\boldsymbol{x})}\boldsymbol{I}_{3\times3}\cdot(\boldsymbol{u}^b - \boldsymbol{u}^a), \end{aligned} \tag{B.10}$$

where the property of the tensor product,

$$(\boldsymbol{u}\otimes\boldsymbol{v})\boldsymbol{w} = (\boldsymbol{w}\cdot\boldsymbol{v})\boldsymbol{u} \tag{B.11}$$

has been used.

B.2.1 Matrix Equations

Also note, the directional derivative for node a is $D\boldsymbol{T}^a(\boldsymbol{x})[\boldsymbol{u}] = -D\boldsymbol{T}^b(\boldsymbol{x})[\boldsymbol{u}]$ since $\boldsymbol{T}^a(\boldsymbol{x}) = -\boldsymbol{T}^b(\boldsymbol{x})$.

Equation (B.10) can be rearranged into matrix form. For a single fiber element e, the

directional derivative in matrix form is

$$D\boldsymbol{T}^{(\mathrm{e})}(\boldsymbol{x}^{(\mathrm{e})})[\boldsymbol{u}^{(\mathrm{e})}] = \boldsymbol{K}^{(\mathrm{e})}\boldsymbol{u}^{(\mathrm{e})} = \begin{bmatrix} \boldsymbol{K}_{aa}^{(\mathrm{e})} & \boldsymbol{K}_{ab}^{(\mathrm{e})} \\ \boldsymbol{K}_{ba}^{(\mathrm{e})} & \boldsymbol{K}_{bb}^{(\mathrm{e})} \end{bmatrix} \begin{bmatrix} \boldsymbol{u}^{a} \\ \boldsymbol{u}^{b} \end{bmatrix}, \tag{B.12}$$

where

$$\boldsymbol{K}_{aa}^{(\mathrm{e})} = \boldsymbol{K}_{bb}^{(\mathrm{e})} = k(\boldsymbol{n}\otimes\boldsymbol{n})_{3x3} + \frac{F}{l(\boldsymbol{x})}\boldsymbol{I}_{3x3}, \quad \text{and} \quad \boldsymbol{K}_{ab}^{(\mathrm{e})} = \boldsymbol{K}_{ba}^{(\mathrm{e})} = -\boldsymbol{K}_{bb}^{(\mathrm{e})}, \tag{B.13}$$

and

$$k \equiv \frac{EAl(\boldsymbol{x})}{L^2}\exp(B\varepsilon) - \frac{F}{l(\boldsymbol{x})}. \tag{B.14}$$

Additionally,

$$(\boldsymbol{n}\otimes\boldsymbol{n})_{3x3} = \begin{bmatrix} \cos(\alpha)\cos(\alpha) & \cos(\alpha)\cos(\beta) & \cos(\alpha)\cos(\gamma) \\ \cos(\beta)\cos(\alpha) & \cos(\beta)\cos(\beta) & \cos(\beta)\cos(\gamma) \\ \cos(\gamma)\cos(\alpha) & \cos(\gamma)\cos(\beta) & \cos(\gamma)\cos(\gamma) \end{bmatrix} \quad \text{and} \quad \mathrm{I}_{3x3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{B.15}$$

with

$$\cos(\alpha) = \frac{x_1^b - x_1^a}{l(\boldsymbol{x})}, \quad \cos(\beta) = \frac{x_2^b - x_2^a}{l(\boldsymbol{x})}, \quad \text{and}\cos(\gamma) = \frac{x_3^b - x_3^a}{l(\boldsymbol{x})} \tag{B.16}$$

and

$$\boldsymbol{u}^a = (x_i^a - x_i^b)\boldsymbol{e}_i \quad \text{and} \quad \boldsymbol{u}^b = (x_i^b - x_i^a)\boldsymbol{e}_i. \tag{B.17}$$

Therefore, $\boldsymbol{u}_a = -\boldsymbol{u}_b$. Using the above relationships, Eq. (B.13) can be written in matrix form as

$$\boldsymbol{K}_{aa}^{(\mathrm{e})} = \begin{bmatrix} k\cos(\alpha)\cos(\alpha) + \frac{F}{l(\boldsymbol{x})} & k\cos(\alpha)\cos(\beta) & k\cos(\alpha)\cos(\gamma) \\ k\cos(\beta)\cos(\alpha) & k\cos(\beta)\cos(\beta) + \frac{F}{l(\boldsymbol{x})} & k\cos(\beta)\cos(\gamma) \\ k\cos(\gamma)\cos(\alpha) & k\cos(\gamma)\cos(\beta) & k\cos(\gamma)\cos(\gamma) + \frac{F}{l(\boldsymbol{x})} \end{bmatrix} \tag{B.18}$$

The Newton-Raphson method for a single fiber element is

$$\boldsymbol{K}^{(\mathrm{e})}\delta\boldsymbol{u}^{(\mathrm{e})} = -\boldsymbol{T}^{(\mathrm{e})}, \text{ where } \boldsymbol{u}^{(\mathrm{e}),t+1} = \boldsymbol{u}^{(\mathrm{e}),t} + \delta\boldsymbol{u}^{(\mathrm{e})}, \tag{B.19}$$

and the superscripts $t+1$ and t denote the current and next iterations, respectively. In

matrix form, $\boldsymbol{u}^{(\mathrm{e})}$ and $\boldsymbol{T}^{(\mathrm{e})}$ are

$$\boldsymbol{u}^{(\mathrm{e})} = \begin{bmatrix} \boldsymbol{u}^a \\ \boldsymbol{u}^b \end{bmatrix}^{(\mathrm{e})} = \begin{bmatrix} x_1^a \\ x_2^a \\ x_3^a \\ x_1^b \\ x_2^b \\ x_3^b \end{bmatrix}^{(\mathrm{e})} \quad \text{and} \quad \boldsymbol{T}^{(\mathrm{e})} = \begin{bmatrix} \boldsymbol{T}^a \\ \boldsymbol{T}^b \end{bmatrix}^{(\mathrm{e})} = F \begin{bmatrix} \frac{(x_1^a - x_1^b)}{l(\boldsymbol{x})} \\ \frac{(x_2^a - x_2^b)}{l(\boldsymbol{x})} \\ \frac{(x_3^a - x_3^b)}{l(\boldsymbol{x})} \\ \frac{(x_1^b - x_1^a)}{l(\boldsymbol{x})} \\ \frac{(x_2^b - x_2^a)}{l(\boldsymbol{x})} \\ \frac{(x_3^b - x_3^a)}{l(\boldsymbol{x})} \end{bmatrix}^{(\mathrm{e})} = F \begin{bmatrix} -\cos(\alpha) \\ -\cos(\beta) \\ -\cos(\gamma) \\ \cos(\alpha) \\ \cos(\beta) \\ \cos(\gamma) \end{bmatrix}^{(\mathrm{e})} \tag{B.20}$$

where the superscript (e) denotes that Eq. (B.20) is for a single fiber element.

APPENDIX C
Multi-scale Coupling Derivation

C.1 Calculation of Volume-Averaged Stress

According to Stylianopoulos and Barocas, the macroscopic stress tensor, S_{ij}, is determined from the forces, T_j^f, that act on the nodes f of a fiber element in the direction j through a volume averaging process given by

$$S_{ij} = \frac{1}{V} \sum_{\text{bcl}} x_i^f T_j^f, \tag{C.1}$$

where the summation is over fiber nodes that lie on the boundary faces of a representative volume element (RVE) and *bcl* stands for boundary cross link. The volume average is over the current volume of the RVE, V, and x_i^f is the i^{th}-component (i=1,2, or 3) of the positions of fiber node f that is on the boundary face of the RVE [48], [101].

C.2 Micro-scale

C.2.1 Cauchy Stress

The macroscopic stress tensor is obtained by applying Gauss's theorem to Eq. (A.55)

$$\begin{aligned}
\sigma_{ij}^{(M)} &= \frac{1}{V^{(m)}} \int_{V^{(m)}} s_{ij}^{(m)} dV^{(m)} \\
&= \frac{1}{V^{(m)}} \int_{V^{(m)}} (s_{kj}^{(m)} x_{i,k}^{(m)}) dV^{(m)} \\
&= \frac{1}{V^{(m)}} \int_{V^{(m)}} (s_{kj}^{(m)} x_i^{(m)})_{,k} dV^{(m)} - \frac{1}{V^{(m)}} \int_{V^{(m)}} s_{kj,k}^{(m)} x_i^{(m)} dV^{(m)} \\
&= \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} n_k s_{kj}^{(m)} x_i^{(m)} dA^{(m)} \\
&= \frac{1}{V^{(m)}} \int_{\partial V^{(m)}} x_i^{(m)} t_j^{(m)} dA^{(m)},
\end{aligned} \tag{C.2}$$

This chapter authored in conjunction with: J. Merson, V. W. L. Chan

where $n_k s_{kj}^{(m)} = t_j^{(m)}$ is the the traction exerted on the boundaries of the RVE. The second line comes from noticing $s_{ij}^{(m)} = s_{kj}^{(m)}\delta_{ki}$, and $x_{k,i} = \delta_{ki}$. The simplification in the fourth line makes use of the microscale momentum balance (e.g. $s_{ij,i} = 0$). In discrete form, the macroscopic stress becomes

$$\sigma_{ij}^{(M)} = \frac{1}{V^{(m)}} \sum_{n \in bcl} {}^{n}x_i^{(m)} {}^{n}T_j^{(m)} \tag{C.3}$$

where ${}^{n}x_i^{(m)}$ is the i^{th} coordinate of the cross-link on the boundary of the RVE (bcl), and ${}^{n}T_j^{(m)}$ is the j^{th} component of the internal force of a cross-link on the boundary.

C.2.2 Cauchy-Stress Derivative

The derivative of the Cauchy stress is evaluated by using Eq. (C.3) and applying the chain rule.

$$\begin{aligned}
\frac{\partial \sigma_{ij}^{(M)}}{\partial u_k^{(M)}} &= \frac{\partial \sigma_{ij}^{(M)}}{\partial {}^{c}u_l^{(m)}} \frac{\partial {}^{c}u_l^{(m)}}{\partial u_k^{(M)}} \\
&= \left[\frac{\partial}{\partial {}^{c}u_l^{(m)}} \left(\frac{1}{V^{(m)}} \right) \sum_{n \in bcl} {}^{n}x_i^{(m)} {}^{n}T_j^{(m)} + \frac{1}{V^{(m)}} \frac{\partial}{\partial {}^{c}u_l^{(m)}} \left(\sum_{n \in bcl} {}^{n}x_i^{(m)} {}^{n}T_j^{(m)} \right) \right] \frac{\partial {}^{c}u_l^{(m)}}{\partial u_k^{(M)}} \\
&= \left[\frac{\partial}{\partial {}^{c}u_l^{(m)}} \left(\frac{1}{V^{(m)}} \right) s_{ij}^{(m)} + \frac{1}{V^{(m)}} \frac{\partial s_{ij}^{(m)}}{\partial {}^{c}u_l^{(m)}} \right] \frac{\partial {}^{c}u_l^{(m)}}{\partial u_k^{(M)}} \\
&= \left[\frac{1}{V^{(m)}} \frac{\partial s_{ij}^{(m)}}{\partial {}^{c}u_l^{(m)}} - \frac{1}{(V^{(m)})^2} \frac{\partial V^{(m)}}{\partial {}^{c}u_l^{(m)}} s_{ij}^{(m)} \right] \frac{\partial {}^{c}u_l^{(m)}}{\partial u_k^{(M)}},
\end{aligned} \tag{C.4}$$

where ${}^{c}u_l^{(m)}$ is the l^{th} corner (left superscript c) displacement at the microscale (right superscript (m)) and $s_{ij}^{(m)} \equiv \sum_{n \in bcl} {}^{n}x_i^{(m)} {}^{n}T_j^{(m)}$. Since our RVE is a hexahedron, the subscript l ranges from 1 to 24; 8 corner nodes $\times$ 3 degrees of freedom per corner node. From Eq. (C.4), the calculation of the stress derivatives involves the calculation of three different derivatives:

$$\frac{\partial s_{ij}^{(m)}}{\partial {}^{c}u_l^{(m)}}, \quad \frac{\partial V^{(m)}}{\partial {}^{c}u_l^{(m)}}, \quad \text{and} \quad \frac{\partial {}^{c}u_l^{(m)}}{\partial u_k^{(M)}}, \tag{C.5}$$

where the first two involve only microscale quantities and the third links the micro and macro scales.

To come up with these derivatives we had to note the equivalence of taking the deriva-

tive with respect to the current coordinate $\boldsymbol{x}$, and the displacement $\boldsymbol{u}$. This is show in the following chain rule:

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{u}}\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{u}}\frac{\partial(\boldsymbol{x}-\boldsymbol{X})}{\partial \boldsymbol{x}} = \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{u}} \tag{C.6}$$

C.2.2.1 Derivative of s_{ij} with Respect to RVE Corner Coordinates

Applying the chain rule, the derivative of s_{ij} can be written as

$$\begin{aligned}
\frac{\partial s_{ij}}{\partial x_l^{\mathrm{RVE}}} &= \frac{\partial s_{ij}}{\partial x_k}\frac{\partial x_k}{\partial x_l^{\mathrm{RVE}}} \\
&= \frac{\partial}{\partial x_k}\left(\sum_{n\in bcl} x_i^n T_j^n\right)\frac{\partial x_k}{\partial x_l^{\mathrm{RVE}}} \\
&= \sum_{n\in bcl}\left(\frac{\partial x_i^n}{\partial x_k}T_j^n + x_i^n\frac{\partial T_j^n}{\partial x_k}\right)\frac{\partial x_k}{\partial x_l^{\mathrm{RVE}}} \\
&= \sum_{n\in bcl}\left(\delta_{ik}T_j^n + x_i^n\frac{\partial T_j^n}{\partial x_k}\right)\frac{\partial x_k}{\partial x_l^{\mathrm{RVE}}}
\end{aligned} \tag{C.7}$$

where x_k is the k^{th} coordinate of the fiber nodes in the RVE; the subscript k ranges from 1 to $N_{dof}^{fn} \equiv 3\ dofs \times$ number of fiber nodes. Since $\frac{\partial T_j^n}{\partial x_k}$ is the $(j,k)^{th}$ element of the tangent-stiffness matrix for cross-links on the boundary, the terms within the parenthesis in Eq. (C.7) are available from the force balance of the fiber network in the RVE.

At this point, we still need to determine the derivative of fiber-node coordinates of the RVE (x_k) with respect to the corner-node coordinates of the RVE (x_l^{RVE}). A relationship between x_k and x_l^{RVE} is obtained by considering how the internal fiber forces (i.e., the residual on fiber network) changes with respect to x_l^{RVE} (i.e., the displacements at the RVE corner nodes):

$$\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}^{RVE}} = \left.\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}^{RVE}}\right|_{\boldsymbol{x}} + \left.\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\right|_{\boldsymbol{x}^{RVE}}\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}^{RVE}}, \tag{C.8}$$

where

$$\boldsymbol{R} \equiv \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{N_{dof}^{fn}} \end{bmatrix}, \quad \boldsymbol{x}^{RVE} \equiv \begin{bmatrix} x_1^{RVE} \\ x_2^{RVE} \\ \vdots \\ x_{24}^{RVE} \end{bmatrix}, \text{ and } \boldsymbol{x} \equiv \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N_{dof}^{fn}} \end{bmatrix}. \tag{C.9}$$

In Eq. (C.8) the change in residual with respect to $\boldsymbol{x}^{RVE}$ has 2 contributions: change due to (1) RVE corner-node displacements when interior-nodes are fixed and (2) interior-node displacements when RVE corner-nodes are fixed. When the forces in the fiber network are balanced, $\partial \boldsymbol{R}/\partial \boldsymbol{x}^{RVE} = 0$, and the derivative $\partial \boldsymbol{x}/\partial \boldsymbol{x}^{RVE}$ needed to evaluate Eq. (C.7) can be determined by:

$$\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}^{RVE}} = -\left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}}\bigg|_{\boldsymbol{x}^{RVE}}\right)^{-1} \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}^{RVE}}\bigg|_{\boldsymbol{x}}. \tag{C.10}$$

As $\partial \boldsymbol{R}/\partial \boldsymbol{x}|_{\boldsymbol{x}^{RVE}}$ is the tangent-stiffness matrix, we are able to determine $\partial \boldsymbol{x}/\partial \boldsymbol{x}^{RVE}$ once $\partial \boldsymbol{R}/\partial \boldsymbol{x}^{RVE}|_{\boldsymbol{x}}$ is known. Note that the inverse of the tangent-stiffness matrix has already been evaluated when performing the Newton-Raphson procedure for balancing the force of the fiber network.

The change of residual with respect to $\boldsymbol{x}^{RVE}$ while the interior nodes are fixed is calculated by a geometric argument. To illustrate this calculation, consider a single fiber node that lies on the boundary of the RVE box:

$$\left[\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{x}^{RVE}}\bigg|_{\boldsymbol{x}}\right] = \begin{bmatrix} \frac{\partial T_1}{\partial x_1^{RVE}} & \frac{\partial T_1}{\partial x_2^{RVE}} & \frac{\partial T_1}{\partial x_3^{RVE}} & \cdots & \frac{\partial T_1}{\partial x_{22}^{RVE}} & \frac{\partial T_1}{\partial x_{23}^{RVE}} & \frac{\partial T_1}{\partial x_{24}^{RVE}} \\ \frac{\partial T_2}{\partial x_1^{RVE}} & \frac{\partial T_2}{\partial x_2^{RVE}} & \frac{\partial T_2}{\partial x_3^{RVE}} & \cdots & \frac{\partial T_2}{\partial x_{22}^{RVE}} & \frac{\partial T_2}{\partial x_{23}^{RVE}} & \frac{\partial T_2}{\partial x_{24}^{RVE}} \\ \frac{\partial T_3}{\partial x_1^{RVE}} & \frac{\partial T_3}{\partial x_2^{RVE}} & \frac{\partial T_3}{\partial x_3^{RVE}} & \cdots & \frac{\partial T_3}{\partial x_{22}^{RVE}} & \frac{\partial T_3}{\partial x_{23}^{RVE}} & \frac{\partial T_3}{\partial x_{24}^{RVE}} \end{bmatrix} \begin{bmatrix} \frac{A_1}{A}\boldsymbol{I}_{3\times 3} \\ \vdots \\ \frac{A_8}{A}\boldsymbol{I}_{3\times 3} \end{bmatrix}, \tag{C.11}$$

where A is the total area of the face on which the fiber node lies and A_i is the portion of the RVE face enclosed by the fiber node and the i^{th} RVE corner node; A_i is nonzero only for fiber nodes that lie on one of the RVE faces. See Fig. C.1 for a schematic of A_i/A, where the blue dot represents a fiber node that lies on an RVE face, the black dots represent the 4 nodes of an RVE face. The total area of the RVE face is $A = A_1 + A_2 + A_3 + A_4$.. To form the full of $\partial \boldsymbol{R}/\partial \boldsymbol{x}^{RVE}|_{\boldsymbol{x}}$, every fiber node must be considered. Since each fiber node contributes three rows to the matrix, the dimensions of $\partial \boldsymbol{R}/\partial \boldsymbol{x}^{RVE}|_{\boldsymbol{x}}$ is $3 \times N_{dof}^{fn}$ by 24 (number of RVE corner nodes $\times$ 3 dofs per corner node).

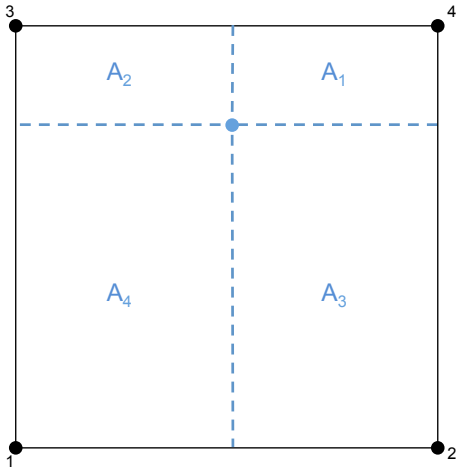

Fig. C.1: Two-dimensional schematic of A_i/A

C.2.2.2 Derivative of V with Respect to RVE Corner Coordinates

The volume can be calculated in the parent domain by

$$V = \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \det\boldsymbol{J}^e d\xi_1 d\xi_2 d\xi_3 = \int_{-1}^{1}\left(\int_{-1}^{1}\left(\int_{-1}^{1} \det\boldsymbol{J}^e d\xi_1\right) d\xi_2\right) d\xi_3, \tag{C.12}$$

which has been written in terms of three consecutive integrals. In the biotissue code, the integrals are evaluated using the Gauss integration formula where Eq. (C.12) becomes

$$V = \sum_{i=1}^{N_{gp}}\sum_{j=1}^{N_{gp}}\sum_{k=1}^{N_{gp}} W_i W_j W_k \det\boldsymbol{J}^e(\xi_1^{(i)}, \xi_2^{(j)}, \xi_3^{(k)}) \tag{C.13}$$

where N_{gp} is the number of Gauss integration points, W_i is the weight of the i^{th} integration point, $\boldsymbol{J}^e$ is the Jacobian matrix, and $\xi_i^{(j)}$ is the i^{th} component of the j^{th} integration point. Note that V is the volume for a hexahedron where $N_{gp} = 2$, $W_i = 1$, and $\xi_i^{(j)} = \pm 1/\sqrt{3}$.

The derivative of the Volume with respect to the RVE corner coordinates can be written in index notation as

$$\begin{aligned}\frac{\partial V}{\partial x_l^{RVE}} &= \int_{-1}^{1}\int_{-1}^{1}\int_{-1}^{1} \frac{\partial}{\partial x_l^{RVE}}\left(\det\boldsymbol{J}^e\right) d\xi_1 d\xi_2 d\xi_3 \\ &= \sum_{i=1}^{N_{gp}}\sum_{j=1}^{N_{gp}}\sum_{k=1}^{N_{gp}} W_i W_j W_k \frac{\partial}{\partial x_l^{RVE}}\left(\det\boldsymbol{J}^e(\xi_1^{(i)}, \xi_2^{(j)}, \xi_3^{(k)})\right),\end{aligned} \tag{C.14}$$

where we have applied the Gauss integration formula (c.f., Eq. (C.13)). The derivative of $\det\boldsymbol{J}^e(\xi_1^{(i)},\xi_2^{(j)},\xi_3^{(k)})$ is

$$\begin{aligned}\frac{\partial}{\partial x_l^{RVE}}\left(\det\boldsymbol{J}^e(\xi_1^{(i)},\xi_2^{(j)},\xi_3^{(k)})\right) = \\ = \frac{\partial}{\partial x_l^{RVE}}\left[x_{,\xi_1}\left(y_{,\xi_2}z_{,\xi_3}-z_{,\xi_2}y_{,\xi_3}\right)-y_{,\xi_1}\left(x_{,\xi_2}z_{,\xi_3}-z_{,\xi_2}x_{,\xi_3}\right)\right. \\ \left.+z_{,\xi_1}\left(x_{,\xi_2}y_{,\xi_3}-y_{,\xi_2}x_{,\xi_3}\right)\right]\Big|_{\xi_1=\xi_1^{(i)},\xi_2=\xi_2^{(j)},\xi_3=\xi_3^{(k)}}. \quad \text{(C.15)}\end{aligned}$$

To illustrate, the x,y,z components of the derivative for the "first" node of the hexahedron $(x_1^{RVE},x_2^{RVE},x_3^{RVE})$ are explicitly written out:

$$\begin{aligned}\frac{\partial\det\boldsymbol{J}^e}{\partial x_1^{RVE}} &= \frac{\partial x_{,\xi_1}}{\partial x_1^{RVE}}\left(y_{,\xi_2}z_{,\xi_3}-z_{,\xi_2}y_{,\xi_3}\right)-y_{,\xi_1}\left(\frac{\partial x_{,\xi_2}}{\partial x_1^{RVE}}z_{,\xi_3}-z_{,\xi_2}\frac{\partial x_{,\xi_3}}{\partial x_1^{RVE}}\right) \\ &\quad + z_{,\xi_1}\left(\frac{\partial x_{,\xi_2}}{\partial x_1^{RVE}}y_{,\xi_3}-y_{,\xi_2}\frac{\partial x_{,\xi_3}}{x_1^{RVE}}\right) \\ &= \det\begin{bmatrix}\frac{\partial x_{,\xi_1}}{\partial x_1^{RVE}} & y_{,\xi_1} & z_{,\xi_1}\\ \frac{\partial x_{,\xi_2}}{\partial x_1^{RVE}} & y_{,\xi_2} & z_{,\xi_2}\\ \frac{\partial x_{,\xi_3}}{\partial x_1^{RVE}} & y_{,\xi_3} & z_{,\xi_3}\end{bmatrix} = \det\begin{bmatrix}\frac{\partial N_1^{6\text{hex}}}{\partial\xi_1} & y_{,\xi_1} & z_{,\xi_1}\\ \frac{\partial N_1^{6\text{hex}}}{\partial\xi_2} & y_{,\xi_2} & z_{,\xi_2}\\ \frac{\partial N_1^{6\text{hex}}}{\partial\xi_3} & y_{,\xi_3} & z_{,\xi_3}\end{bmatrix} \\ \frac{\partial\det\boldsymbol{J}^e}{\partial x_2^{RVE}} &= x_{,\xi_1}\left(\frac{\partial y_{,\xi_2}}{\partial x_2^{RVE}}z_{,\xi_3}-z_{,\xi_2}\frac{\partial y_{,\xi_3}}{\partial x_2^{RVE}}\right)-\frac{\partial y_{,\xi_1}}{\partial x_2^{RVE}}\left(x_{,\xi_2}z_{,\xi_3}-z_{,\xi_2}x_{,\xi_3}\right) \\ &\quad + z_{,\xi_1}\left(x_{,\xi_2}\frac{\partial y_{,\xi_3}}{\partial x_2^{RVE}}-\frac{\partial y_{,\xi_2}}{\partial x_2^{RVE}}x_{,\xi_3}\right) \\ &= \det\begin{bmatrix}x_{,\xi_1} & \frac{\partial y_{,\xi_1}}{\partial x_2^{RVE}} & z_{,\xi_1}\\ x_{,\xi_2} & \frac{\partial y_{,\xi_2}}{\partial x_2^{RVE}} & z_{,\xi_2}\\ x_{,\xi_3} & \frac{\partial y_{,\xi_3}}{\partial x_2^{RVE}} & z_{,\xi_3}\end{bmatrix} = \det\begin{bmatrix}x_{,\xi_1} & \frac{\partial N_2^{6\text{hex}}}{\partial\xi_1} & z_{,\xi_1}\\ x_{,\xi_2} & \frac{\partial N_2^{6\text{hex}}}{\partial\xi_2} & z_{,\xi_2}\\ x_{,\xi_3} & \frac{\partial N_2^{6\text{hex}}}{\partial\xi_3} & z_{,\xi_3}\end{bmatrix} \\ \frac{\partial\det\boldsymbol{J}^e}{\partial x_3^{RVE}} &= x_{,\xi_1}\left(y_{,\xi_2}\frac{\partial z_{,\xi_3}}{\partial x_3^{RVE}}-\frac{\partial z_{,\xi_2}}{\partial x_3^{RVE}}y_{,\xi_3}\right)-y_{,\xi_1}\left(x_{,\xi_2}\frac{\partial z_{,\xi_3}}{\partial x_3^{RVE}}-\frac{\partial z_{,\xi_2}}{\partial x_3^{RVE}}x_{,\xi_3}\right) \\ &\quad + \frac{\partial z_{,\xi_1}}{\partial x_3^{RVE}}\left(x_{,\xi_2}y_{,\xi_3}-y_{,\xi_2}x_{,\xi_3}\right) \\ &= \det\begin{bmatrix}x_{,\xi_1} & y_{,\xi_1} & \frac{\partial z_{,\xi_1}}{\partial x_2^{RVE}}\\ x_{,\xi_2} & y_{,\xi_2} & \frac{\partial z_{,\xi_2}}{\partial x_2^{RVE}}\\ x_{,\xi_3} & y_{,\xi_3} & \frac{\partial z_{,\xi_3}}{\partial x_2^{RVE}}\end{bmatrix} = \det\begin{bmatrix}x_{,\xi_1} & z_{,\xi_1} & \frac{\partial N_3^{6\text{hex}}}{\partial\xi_1}\\ x_{,\xi_2} & z_{,\xi_2} & \frac{\partial N_3^{6\text{hex}}}{\partial\xi_2}\\ x_{,\xi_3} & z_{,\xi_3} & \frac{\partial N_3^{6\text{hex}}}{\partial\xi_3}\end{bmatrix}. \quad \text{(C.16)}\end{aligned}$$

Note that the formula for the derivatives for each component of the "second" to

"eighth" nodes of the hexahedron are essentially the same, with the only difference being the shape function: $N_l^{6\text{hex}}$ with $l = 4$ to 24 for components corresponding to the second to eighth nodes. The relationship $\partial x_{,\xi_i}/\partial x_j^{RVE} = \partial N_j^{6\text{hex}}/\partial \xi_i$ was used in Eq. (C.16).

C.2.2.3 Derivative of RVE Corner Coordinates with Respect to Finite-Element Displacements

The matrix $[\partial \boldsymbol{x}^{RVE}/\partial \boldsymbol{u}^k]$ has dimensions of 24 (8 RVE corner nodes $\times$ 3 dofs per node) $\times$ 12 (4 FE tetrahedron nodes $\times$ 3 dofs per node) for each element:

$$\left[\frac{\partial \boldsymbol{x}^{RVE}}{\partial \boldsymbol{u}^k}\right] = \begin{bmatrix} \frac{\partial x_1^{RVE}}{\partial u_1^k} & \frac{\partial x_1^{RVE}}{\partial u_2^k} & \frac{\partial x_1^{RVE}}{\partial u_3^k} & \cdots & \frac{\partial x_1^{RVE}}{\partial u_{10}^k} & \frac{\partial x_1^{RVE}}{\partial u_{11}^k} & \frac{\partial x_1^{RVE}}{\partial u_{12}^k} \\ \frac{\partial x_2^{RVE}}{\partial u_1^k} & \frac{\partial x_2^{RVE}}{\partial u_2^k} & \frac{\partial x_2^{RVE}}{\partial u_3^k} & & \frac{\partial x_2^{RVE}}{\partial u_{10}^k} & \frac{\partial x_2^{RVE}}{\partial u_{11}^k} & \frac{\partial x_2^{RVE}}{\partial u_{12}^k} \\ \frac{\partial x_3^{RVE}}{\partial u_1^k} & \frac{\partial x_3^{RVE}}{\partial u_2^k} & \frac{\partial x_3^{RVE}}{\partial u_3^k} & & \frac{\partial x_3^{RVE}}{\partial u_{10}^k} & \frac{\partial x_3^{RVE}}{\partial u_{11}^k} & \frac{\partial x_3^{RVE}}{\partial u_{12}^k} \\ & \vdots & & & & \vdots & \\ \frac{\partial x_{22}^{RVE}}{\partial u_1^k} & \frac{\partial x_{22}^{RVE}}{\partial u_2^k} & \frac{\partial x_{22}^{RVE}}{\partial u_3^k} & & \frac{\partial x_{22}^{RVE}}{\partial u_{10}^k} & \frac{\partial x_{22}^{RVE}}{\partial u_{11}^k} & \frac{\partial x_{22}^{RVE}}{\partial u_{12}^k} \\ \frac{\partial x_{23}^{RVE}}{\partial u_1^k} & \frac{\partial x_{23}^{RVE}}{\partial u_2^k} & \frac{\partial x_{23}^{RVE}}{\partial u_3^k} & \cdots & \frac{\partial x_{23}^{RVE}}{\partial u_{10}^k} & \frac{\partial x_{23}^{RVE}}{\partial u_{11}^k} & \frac{\partial x_{23}^{RVE}}{\partial u_{12}^k} \\ \frac{\partial x_{24}^{RVE}}{\partial u_1^k} & \frac{\partial x_{24}^{RVE}}{\partial u_2^k} & \frac{\partial x_{24}^{RVE}}{\partial u_3^k} & & \frac{\partial x_{24}^{RVE}}{\partial u_{10}^k} & \frac{\partial x_{24}^{RVE}}{\partial u_{11}^k} & \frac{\partial x_{24}^{RVE}}{\partial u_{12}^k} \end{bmatrix}$$

$$= \begin{bmatrix} \Delta N_1^{(1)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(1)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(1)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(1)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(2)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(2)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(2)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(2)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(3)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(3)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(3)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(3)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(4)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(4)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(4)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(4)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(5)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(5)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(5)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(5)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(6)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(6)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(6)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(6)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(7)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(7)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(7)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(7)} \boldsymbol{I}_{3\times3} \\ \Delta N_1^{(8)} \boldsymbol{I}_{3\times3} & \Delta N_2^{(8)} \boldsymbol{I}_{3\times3} & \Delta N_3^{(8)} \boldsymbol{I}_{3\times3} & \Delta N_4^{(8)} \boldsymbol{I}_{3\times3} \end{bmatrix}, \tag{C.17}$$

where

$$\begin{aligned} \Delta N_j^{(k)} \boldsymbol{I}_{3\times3} &\equiv \begin{bmatrix} \Delta N_j^{(k)} & 0 & 0 \\ 0 & \Delta N_j^{(k)} & 0 \\ 0 & 0 & \Delta N_j^{(k)} \end{bmatrix} \\ \Delta N_j^{(k)} &\equiv N_j^{4\text{tet}}(\xi_1^k, \xi_2^k, \xi_3^k) - N_j^{4\text{tet}}(\xi_1^{gp}, \xi_2^{gp}, \xi_3^{gp}). \end{aligned} \tag{C.18}$$

In the matrix of Eq. (C.17), each 3×3 submatrix describes the relationship between the degrees of freedom of an RVE corner node and an FE node. These relationships are approximated as 3×3 diagonal matrices where the diagonal terms are calculated as the difference between the j^{th} (j ranges from 1 to 4 for tetrahedron) shape function at the barycentric coordinate of the k^{th} RVE corner-node and that at the barycentric coordinate of the Gauss integration point (c.f., Eq. (C.18)). The implicit assumption in this method is that the difference in scale between the RVE and FE entity is substantial.

C.3 Derivative of Macroscopic Stress With Respect to FE Node Positions

As discussed in Section C.1, the macroscale stresses are calculated from the equilibrium fiber-node positions of each fiber element within a RVE. Since the macroscale stresses explicitly depend on the fiber-node positions, a natural first step is to consider the derivative of the macroscopic stress with respect to the fiber-node positions.

The directional derivative of S_{ij} is

$$\begin{aligned} DS_{ij}[\boldsymbol{u}] &= D\left(\frac{1}{V}\sum_{\text{bcl}} x_i^f T_j^f\right)[\boldsymbol{u}] \\ &= D\left(\frac{1}{V}\right)[\boldsymbol{u}]\sum_{\text{bcl}} x_i^f T_j^f + \frac{1}{V}\sum_{\text{bcl}}\left[x_i^f DT_j^f[\boldsymbol{u}] + Dx_i^f[\boldsymbol{u}]T_j^f\right] \\ &= D\left(\frac{1}{V}\right)[\boldsymbol{u}]VS_{ij} + \frac{1}{V}\sum_{\text{bcl}} C_{ij}^f, \end{aligned} \tag{C.19}$$

where

$$\begin{aligned} C_{ij}^f &\equiv x_i^f DT_j^f[\boldsymbol{u}] + Dx_i^f[\boldsymbol{u}]T_j^f \\ &= x_i^f \frac{\partial T_j^f}{\partial x_k^f}u_k^f + \frac{\partial x_i^f}{\partial x_k^f}u_k^f T_j^f, \end{aligned} \tag{C.20}$$

where C_{ij}^f is for a single fiber node (as indicated by the superscript f). The summation in Eq. (C.19) account for all fiber nodes that lie on the RVE boundary. As seen in Eq. (C.19), the directional derivative of S_{ij} requires the calculation of C_{ij}^f and the directional derivative of $1/V$.

C.3.1 Calculation of C_{ij}^f

The term C_{ij}^f in Eq. (C.20) can be explicitly written out as

$$\begin{aligned}\left[C_{ij}^f\right] &= \begin{bmatrix} x_1^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + \frac{\partial x_1^f}{\partial x_k^f} u_k^f T_1^f & x_1^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + \frac{\partial x_1^f}{\partial x_k^f} u_k^f T_2^f & x_1^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + \frac{\partial x_1^f}{\partial x_k^f} u_k^f T_3^f \\ x_2^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + \frac{\partial x_2^f}{\partial x_k^f} u_k^f T_1^f & x_2^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + \frac{\partial x_2^f}{\partial x_k^f} u_k^f T_2^f & x_2^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + \frac{\partial x_2^f}{\partial x_k^f} u_k^f T_3^f \\ x_3^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + \frac{\partial x_3^f}{\partial x_k^f} u_k^f T_1^f & x_3^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + \frac{\partial x_3^f}{\partial x_k^f} u_k^f T_2^f & x_3^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + \frac{\partial x_3^f}{\partial x_k^f} u_k^f T_3^f \end{bmatrix} \\ &= \begin{bmatrix} x_1^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + u_1^f T_1^f & x_1^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + u_1^f T_2^f & x_1^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + u_1^f T_3^f \\ x_2^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + u_2^f T_1^f & x_2^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + u_2^f T_2^f & x_2^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + u_2^f T_3^f \\ x_3^f \frac{\partial T_1^f}{\partial x_k^f} u_k^f + u_3^f T_1^f & x_3^f \frac{\partial T_2^f}{\partial x_k^f} u_k^f + u_3^f T_2^f & x_3^f \frac{\partial T_3^f}{\partial x_k^f} u_k^f + u_3^f F_3^f \end{bmatrix}, \end{aligned} \tag{C.21}$$

where

$$T_1^a \equiv -F\cos(\alpha), \quad T_2^a \equiv -F\cos(\beta), T_3^a \equiv -F\cos(\gamma), \quad \text{and} \quad T_i^a = -T_i^b, \tag{C.22}$$

and the second equality in Eq. (C.21) uses the relationship

$$\frac{\partial x_i^f}{\partial x_k^f} = \delta_{ik}. \tag{C.23}$$

Since $C_{ij}^f = C_{ji}^f$ in Eq. (C.21), only the six terms on the upper triangle of the matrix are unique. Consequently, the matrix in Eq. (C.21) can be rewritten as a matrix-vector product

$$\begin{bmatrix} C_{11}^f \\ C_{12}^f \\ C_{13}^f \\ C_{22}^f \\ C_{23}^f \\ C_{33}^f \end{bmatrix} = \begin{bmatrix} x_1^f \frac{\partial T_1^f}{\partial x_1^f} + T_1^f & x_1^f \frac{\partial T_1^f}{x_2^f} & x_1^f \frac{\partial T_1^f}{x_3^f} \\ x_1^f \frac{\partial T_2^f}{\partial x_1^f} + T_2^f & x_1^f \frac{\partial T_2^f}{x_2^f} & x_1^f \frac{\partial T_2^f}{x_3^f} \\ x_1^f \frac{\partial T_3^f}{\partial x_1^f} + T_3^f & x_1^f \frac{\partial T_3^f}{x_2^f} & x_1^f \frac{\partial T_3^f}{x_3^f} \\ x_2^f \frac{\partial T_2^f}{\partial x_1^f} & x_2^f \frac{\partial T_2^f}{x_2^f} + T_2^f & x_2^f \frac{\partial T_2^f}{x_3^f} \\ x_2^f \frac{\partial T_3^f}{\partial x_1^f} & x_2^f \frac{\partial T_3^f}{x_2^f} + T_3^f & x_2^f \frac{\partial T_3^f}{x_3^f} \\ x_3^f \frac{\partial T_3^f}{\partial x_1^f} & x_3^f \frac{\partial T_3^f}{x_2^f} & x_3 \frac{\partial T_3^f}{x_3^f} + T_3^f \end{bmatrix} \begin{bmatrix} u_1^f \\ u_2^f \\ u_3^f \end{bmatrix}. \tag{C.24}$$

Furthermore, since $C_{ij}^f = C_{ji}^f$, the off diagonal terms (i.e., C_{12}^f, C_{13}^f, and C_{23}^f) can be expressed

as averages:

$$\begin{bmatrix} x_1^f \frac{\partial T_1^f}{x_1^f} + T_1^f & x_1^f \frac{\partial T_1^f}{x_2^f} & x_1^f \frac{\partial T_1^f}{x_3^f} \\ \frac{1}{2}\left(x_1^f \frac{\partial T_2^f}{x_1^f} + x_2^f \frac{\partial T_1^f}{\partial x_1^f} + T_2^f\right) & \frac{1}{2}\left(x_1^f \frac{\partial T_2^f}{x_2^f} + x_2^f \frac{\partial T_1^f}{\partial x_2^f} + T_1^f\right) & \frac{1}{2}\left(x_1^f \frac{\partial T_2^f}{x_3^f} + x_2^f \frac{\partial T_1^f}{\partial x_3^f}\right) \\ \frac{1}{2}\left(x_1^f \frac{\partial T_3^f}{x_1^f} + x_3^f \frac{\partial T_1^f}{\partial x_1^f} + T_3^f\right) & \frac{1}{2}\left(x_1^f \frac{\partial T_3^f}{x_2^f} + x_3^f \frac{\partial T_1^f}{x_2^f}\right) & \frac{1}{2}\left(x_1^f \frac{\partial T_3^f}{x_3^f} + x_3^f \frac{\partial T_1^f}{x_3^f} + T_1^f\right) \\ x_2^f \frac{\partial T_2^f}{x_1^f} & x_2^f \frac{\partial T_2^f}{x_2^f} + T_2^f & x_2^f \frac{\partial T_2^f}{x_3^f} \\ \frac{1}{2}\left(x_2^f \frac{\partial T_3^f}{x_1^f} + x_3^f \frac{\partial T_2^f}{x_1^f}\right) & \frac{1}{2}\left(x_2^f \frac{\partial T_3^f}{x_2^f} + x_3^f \frac{\partial T_2^f}{x_2^f} + T_3^f\right) & \frac{1}{2}\left(x_2^f \frac{\partial T_3^f}{x_3^f} + x_3^f \frac{\partial T_2^f}{x_3^f} + T_2^f\right) \\ x_3^f \frac{\partial T_3^f}{x_1^f} & x_3^f \frac{\partial T_3^f}{x_2^f} & x_3^f \frac{\partial T_3^f}{x_3^f} + T_3^f \end{bmatrix} \begin{bmatrix} u_1^f \\ u_2^f \\ u_3^f \end{bmatrix} \tag{C.25}$$

Note that $\partial T_i^f / \partial x_j^f$ is the i, j^{th} element of Eq. (B.18) when $f = a$. Note that the matrix explicitly written out in Eq. (C.25) constitutes a single fiber node of the variable ttdSdy. The full ttdSdy variable, which is of size 6 $\times$ num_dof, is constructed when summing over all boundary nodes (see summation in Eq. (C.19)).

C.3.2 Directional Derivative of $1/V$

In order to determine the directional derivative of S_{ij} in Eq. (C.19), we need to also determine $D\left(1/V\right)[\boldsymbol{u}]$:

$$D\frac{1}{V}[\boldsymbol{u}] = \frac{\partial(1/V)}{\partial x_i^f} u_i^f = -\frac{1}{V^2}\left(\frac{\partial V}{\partial x_i^f} u_i^f\right). \tag{C.26}$$

Equation (C.26) can be further written in matrix form as:

$$D\frac{1}{V}[\boldsymbol{u}] = -\frac{1}{V^2}\begin{bmatrix} \frac{\partial(V)}{\partial x_1^f} & \frac{\partial(V)}{\partial x_2^f} & \frac{\partial(V)}{\partial x_3^f} \end{bmatrix} \begin{bmatrix} u_1^f \\ u_2^f \\ u_3^f \end{bmatrix}. \tag{C.27}$$

C.3.3 Combining Everything Together

Substituting Eq. (C.26) in Eq. (C.19) we obtain

$$\begin{aligned} DS_{ij}[\boldsymbol{u}] &= \frac{1}{V}\sum_{\text{bcl}}\left(x_i^f\frac{\partial T_j^f}{\partial x_k^f}u_k^f + \frac{\partial x_i^f}{\partial x_k^f}u_k^f T_j^f\right) - \frac{1}{V^2}\left(\frac{\partial V}{\partial x_k^f}u_k^f\right)VS_{ij} \\ &= \frac{1}{V}\left[\sum_{\text{bcl}}\left(x_i^f\frac{\partial T_j^f}{\partial x_k^f} + \frac{\partial x_i^f}{\partial x_k^f}T_j^f\right) - \frac{\partial V}{\partial x_k^f}S_{ij}\right]u_k^f \end{aligned} \tag{C.28}$$