



Summary Report for the Human Electro-Muscular Incapacitation (HEMI) Weapon Effects Modeling and Simulation System

31 December 2021

Prepared for:
Joint Intermediate Force Capabilities Office (JIFCO)
3097 Range Road
Quantico, VA 22134-5100
c/o Dr. Shannon Foley

Prepared by:
Applied Research Associates, Inc.
Arlington Division
801 N. Quincy Street, Suite 700
Arlington, VA 22203

Authors: Timothy Dennis, Kidist Maxwell, Camille Romanowski, and
Rodney Metoyer

Reviewer: Stephen Beaulieu

Contract: N00174-17-D-0032
Delivery Order: N0017419F0504

Distribution Statement A:

Unclassified; Distribution Unlimited

Table of Contents

1. Introduction	2
1.1. HEMI and HEMI Weapons	3
1.2. Military Need for HEMI M&S	4
2. HEMI Mechanisms	6
2.1. Bioelectromagnetic Phenomena	7
2.2. Physiological Phenomena	8
2.2.1. Incapacitation (The Intended Effect)	8
2.2.2. Other Physiological Mediators (Unintended Effects)	10
3. Mathematical Models	14
3.1. Bioelectromagnetic Models	14
3.2. Renal Injury Model	14
3.3. Muscle Physiology Model	14
3.3.1. Neuromuscular Junction	15
3.3.2. Calcium Dynamics of Excitation Contraction Coupling	15
3.3.3. Muscle Force	19
4. HEMI M&S System Development Approach	19
4.1. Common Data Model	20
4.2. Configuration Management	21
4.3. Development Philosophy	21
5. Software Architecture	22
5.1. Nerve Module	22
5.1.1. Overview	22
5.1.2. SENN Nerve Model	25
5.2. Voltage Module	26
5.2.1. Overview	27
5.2.2. SENN Native Voltage Models	28
5.2.3. FDTD Voltage Model	30
5.3. Muscle Module	33
5.3.1. Overview	33
5.3.2. Wexler Muscle Module	34

5.4. Execution Control	35
6. Verification and Validation.....	37
6.1. Verification	37
6.2. Validation.....	37
7. Outcomes Quantification.....	38
7.1. Traditional Statistical Approach	39
7.2. Modern Causal Inference.....	41
7.2.1. Structural Casual Modeling	42
8. Summary, Conclusions, and Possible Next Steps.....	42
8.1. Near Future Objectives	43
8.1.1. Continue the Computational Modeling Effort	43
8.1.2. Formal Structural Modeling for Causal Inference	43
8.1.3. Collaborate with Empiricists.....	44
9. References	44
Appendix A. V&V Report	50
A.1. SENN Nerve Model Verification.....	50
A.1.1. Results of Nerve Verification Case 1.....	51
A.1.2. Nerve Verification Results Conclusions	52
A.2. Muscle Model Verification	53
A.2.1. Muscle Verification Test 1.....	53
A.2.2. Muscle Verification Test 2.....	55
A.2.3. Muscle Verification Test 3.....	56
A.3. Conclusion	58
Appendix B. HEMI User Guide and Training Manual	59
B.1. Building the HEMI Library and its Dependencies	59
B.2. Building the Dependencies	59
B.3. Building the HEMI Library	60
B.4. Running hemi-example.exe	60
B.5. Implementation of the hemi-example sample.....	62
Appendix C. Model Equations and Symbols	64
C.1. Neuromuscular Junction Model Parameters and Variables	64
C.2. Muscle Model Parameters and Variables	64

Appendix D.	FDTD Voltage.....	66
--------------------	--------------------------	-----------

List of Figures

Figure 1. General approach for the HEMI physiological modeling project.	2
Figure 2. Top-level causal diagram of HEMI weapon events.....	3
Figure 3. Mechanisms of HEMI exposure outcomes.	6
Figure 4. Example of a bioelectromagnetic process. An electromagnetic field causes a local ion current and subsequent action potential.	7
Figure 5. Motor units and motor neuron pools (Boron 2012).....	8
Figure 6. Synaptic transmission at the neuromuscular junction, reproduced from Figure 8-2 on page 215 in (Boron 2012).	9
Figure 7. Excitation-contraction coupling in skeletal muscle, reproduced from Figure 9-3 on page 240 in (Boron 2012).	10
Figure 8. General overview of the calcium cycle in skeletal muscle, reprinted from (Berchtold, Brinkmeier, and Müntener 2000).	12
Figure 9. Conceptual overview of the relation between calcium and ATP dynamics.	12
Figure 10. The “Vicious cycle” model of how increased influx of Ca^{2+} in the myoplasm can induce skeletal muscle damage, reprinted from Figure 2 of (Hanne Gissel 2005).	14
Figure 11. Calcium pump activity fraction vs. concentration of available ATP.....	18
Figure 12. Calcium pump rate vs. proximal calcium concentration at different [ATP] values.	18
Figure 13. High level operationl viewpoint of the HEMI M&S system software architecture. SAM and MRM are future modules.	20
Figure 14. Screenshot of a git history graph for the HEMI M&S System.	21
Figure 15. Causal complexity of rhabdomyolysis (Pirone 2017).	39
Figure 16. An extension of Figure 2 including exogenous variables.	41
Figure 17. A notional causal diagram for rhabdomyolysis and acute renal injury.....	42
Figure 18. Transmembrane potential of select nodes for Test Case 1.	51
Figure 19. Transmembrane voltage of the print node from the C++ and Fortran implementations for Test Case 1 (left) and the percent difference between them (right).	52
Figure 20. Transmembrane voltage of the highest-voltage node from the C++ and Fortran implementations for Test Case 1 (left) and the percent difference between them (right).	52
Figure 21. Figure 3A from (Otazu, Futami, and Hoshimiya 2001).	54
Figure 22. Distal calcium concentration results from the first verification test.....	54

Figure 23. Figure 3B from (Otazu, Futami, and Hoshimiya 2001).	55
Figure 24. Distal calcium concentration results from the second verification test.	56
Figure 25. Inactivation site bound fraction results from the second verification test.	56
Figure 26. Figure 6A from (Otazu, Futami, and Hoshimiya 2001).	57
Figure 27. Distal calcium concentration results from the third verification test.	57
Figure 28. Inactivation site bound fraction results from the third verification test.	58
Figure 29. Distal calcium concentration (left) and pump-adjacent ATP concentration (right) during end-to-end simulation.	58

List of Tables

Table 1. Test Case 1 input parameters.....	50
Table 2. Test Case 1 maximum percent difference between Fortran and C++ implementations.	51
Table 3. Nerve model verification summary data.....	53
Table 4. Distal calcium concentrations at select times during simulation.	54
Table 5. Muscle model states.....	64
Table 6. Muscle model states.....	65
Table 7. Muscle model state- and time-dependent rates	65
Table 8. Muscle model parameters.....	66

Mechanistic Modeling of the Effects of Human Electromuscular Incapacitation (HEMI)

Scientists and engineers at Applied Research Associates, Inc. (ARA) are developing an end-to-end composite model of the physiological effects of HEMI weapons and the risk of significant injury. The model begins with the generation of electromagnetic fields and current in the body and follows the causal chains through to the intended and unintended outcomes. This report describes how we have integrated two models of the electromagnetic state in the body during HEMI weapon exposure with an existing nerve model and a modified muscle model towards prediction of rhabdomyolysis and acute kidney injury. Additionally, this report describes how the extensible software infrastructure that was built to integrate the models provides added value to the government. We show that ARA is developing more than a model, we are building a modeling capability.

The HEMI M&S system is more than a model. It is a modeling capability.

To support the DoD acquisition process and requirements, a modeling solution must enable estimation of the risk of injury and effectiveness of novel HEMI sources and exposure scenarios, necessarily involving the modeling of phenomena that are not currently characterized. Consequently, in the composite model system, some constituent models may need to evolve, and others may need to be replaced as new knowledge emerges. ARA recognized that we could meet this need by providing the government an innovative modeling capability – software infrastructure that supports a hierarchical modeling approach.

This report describes work done in the most recent phase of an ongoing project. We began this project with two fundamental assumptions which lead us to the conclusion above.

- 1) The specific needs of the JIFCO customers who will be using the HEMI M&S system for acquisition and other military purposes are likely to evolve.
- 2) The body of knowledge of the physiological effects of HEMI weapon exposure is incomplete and therefore likely to grow over time.

From these assumptions, we determined that any model of the physiological effects of HEMI weapons must be modifiable and extensible. Furthermore, even if the model is modifiable and extensible, it is possible for the needs of the user to evolve beyond the utility of the model. Therefore, any software infrastructure built to construct and implement the model must, itself, be modifiable and extensible to facilitate both the extension of existing models and the development of new models. This idea is a key reason why ARA's HEMI physiological modeling solution is more valuable to the DoD than separate, standalone models of HEMI exposure from known sources.

In this report, we describe the status the HEMI M&S system. We describe why this system infrastructure is a crucial aspect to success of this effort, providing a sound foundation which may be used by ARA and other investigators and organizations to extend the existing models and to develop new capabilities to meet evolving DoD requirements. The physiological models developed in this phase are also described in detail, and the M&S system is exercised with input from existing HEMI sources.

This Executive Summary is followed by Section 1, an introduction to the effort. Section 2 provides an overview of the causal chains of HEMI, beginning with exposure to a HEMI weapon and ending with the intended and/or unintended outcomes. Then, in Section 3, the mathematical models describing the HEMI physiological processes are described and discussed. Next, Section 4 provides an overview of our development approach which is driven by our desire to construct robust software infrastructure to facilitate interorganizational development of software tools that meet JIFCO and other military-specific needs as requirements mature and as knowledge of the biological mechanisms of HEMI outcomes evolves. Then, in Section 5 we describe the software architecture and specific aspects of the implementation in detail. Next, Section 6 provides a brief V&V update (with more detail provided in the V&V report that is included as Appendix A). Finally, an outcomes quantification approach which includes novel methods of causal inference is discussed in Section 7, and concluding remarks are given in Section 8.

1. Introduction

Human Electro-Muscular Incapacitation (HEMI) weapons are developed with the intent to incapacitate a subject without being lethal. However, under some use cases, these weapons have been linked to serious injury and even death (Cooks et al. 2016b; 2016a; Dawes et al. 2011; Gleason 2015; Ho et al. 2011; Sanford et al. 2011). ARA is developing a generalized computational model, based on underlying physical and biological processes, which will use device characteristics and potential exposure scenarios as inputs to estimate the risk of significant injury following application of a HEMI weapon. The long-term goal of this research is to develop a tool that estimates the efficacy and risk of injury under different use scenarios for devices with different characteristics. This goal is being accomplished through mathematical modeling of biophysical and physiological responses for a given HEMI exposure. The model results are then employed to quantify the probability of occurrence of events, including specific injuries and the overall risk of significant injury (Figure 1).

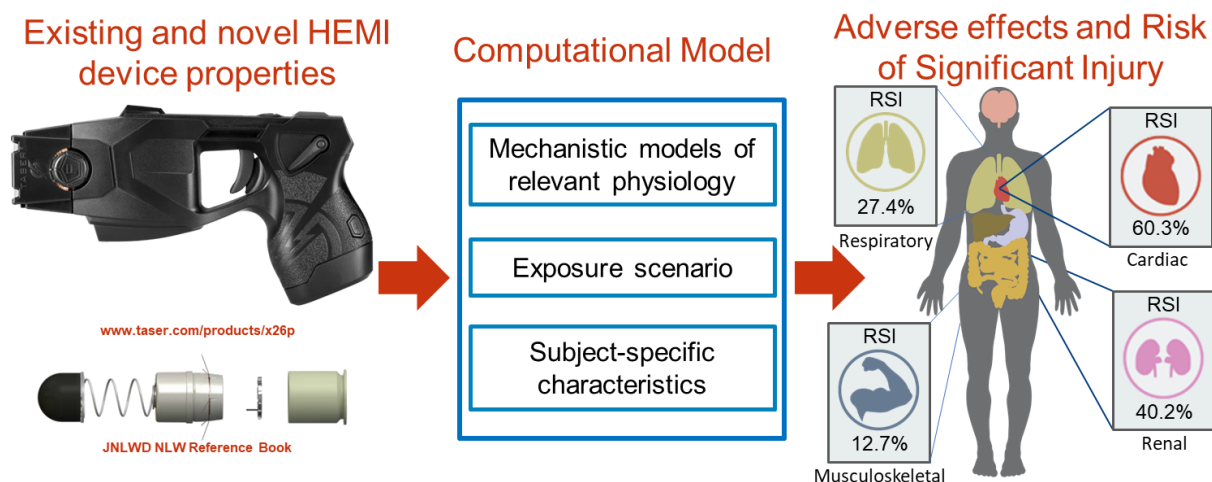


Figure 1. General approach for the HEMI physiological modeling project.

The two paramount features of the ARA HEMI M&S software system under development are 1) modularity and 2) extensibility. Modularity is critical because it makes possible the substitution of models and sub-models. For example, one nerve model may be replaced with another if, in the

future, an existing model is found to be lacking given some criteria. Extensibility is important to support elasticity in the acquisition application – where the use-case definitions and modeling objectives may evolve and expand over time. Furthermore, new knowledge and data may alter conceptual and mathematical models, necessitating an extension of the computational models.

To achieve modularity, well-defined boundaries are required; in other words, an explicit definition of the input and output data is needed. These data may be defined ad hoc, but a more robust methodology for defining model boundaries is in terms of the causality of events (Pearl and Mackenzie 2018), providing a necessary basis for determining conceptual interoperability. Likewise, extensibility requires an explicit definition of data structures for exchange between modeling elements. This is particularly important when using composite models like those that make up the current HEMI physiological effects model. Thus, a common data model (CDM) is necessary for modular and extensible software, and an explicit description of causality contributes to interoperability. Another advantage to the explicit representation of causal assumptions is that it enables quantitative causal inference. Accordingly, a CDM for HEMI-related physiologic data forms the foundation of the HEMI M&S software architecture (discussed in detail in Section 4), and we have begun the work of recording causal assumptions (discussed in Section 7).

1.1. HEMI and HEMI Weapons

Human electro-muscular incapacitation (HEMI) is the desired incapacitation event that is caused by application of a HEMI weapon. Conducted energy weapons (CEWs) are widely used to achieve HEMI. The term “HEMI weapon” is often used interchangeably with CEW, including in this report.

The intended effect of a HEMI weapon is the denial of volitional control of the body without causing significant injury or death. However, unintended effects such as ocular injury, seizures, ventricular fibrillation, falls, and other injuries have been associated with HEMI exposures (Maier, Nance, Price, Sherry, Reilly, et al. 2005; Maier, Nance, Price, Sherry, and Reilly 2005; Pasquier et al. 2011; Park et al. 2013; Zipes 2014; Gleason 2015; Gross, Porterieko, and D’Andrea Joseph 2013; Tyagi, Gill, and Felton 2017). The intended outcome (incapacitation) and the unintended outcomes (significant injuries) are not mutually exclusive. A simple causal diagram of CEW exposure is shown in Figure 2.

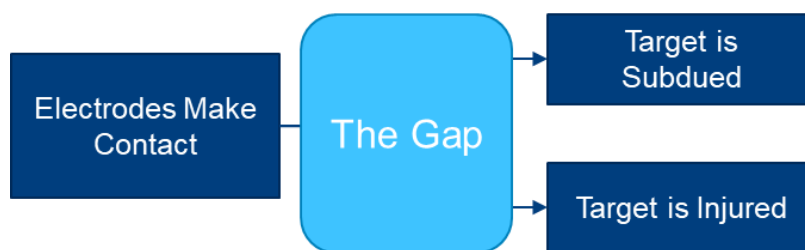


Figure 2. Top-level causal diagram of HEMI weapon events

The light-blue box in the center of Figure 2 represents the physiological mechanisms of the outcomes. Some of these mechanisms are well studied; for example, a rich body of literature may be found describing the development of electromagnetic fields in the body and the

subsequent activation of excitable tissues¹, particularly for motor neurons. In fact, high-quality commercial off-the-shelf software² may be purchased to accurately estimate excitation of tissues under a wide range of conditions.

However, although the mechanisms of nerve excitation by external application of electrical energy are well studied, the subsequent mechanisms that lead to the outcomes of interest (efficacy, injury) are not well understood. For example, recent evidence suggests that incapacitation is achieved through activation of spinal reflex arcs rather than by competitive activation of motor neurons (Despa et al. 2009). A more detailed discussion of this mechanism is provided in Section 2.2.1. Considering the complexity of effects such as rhabdomyolysis, a traditional FEM approach to modeling that only includes the activation of excitable tissues is insufficient. A systems approach is warranted, e.g., systems biology (Kohl et al. 2010). Our approach to identify the mechanisms and fill the gap in Figure 2 is based on principles systems biology.

1.2. Military Need for HEMI M&S

Effectiveness and safety ranges for traditional HEMI devices are well established. The conventional hand-held CEWs used by law enforcement have been employed millions of times, (Kroll et al. 2019), and though there is some controversy surrounding the actual frequency of fatalities and injuries due to methodologic limitations, including heavy involvement in the research by weapon manufacturers (Baliatsas et al. 2021), the risk of injury is undeniably low. However, as previously mentioned, a complete understanding of the biomechanisms involved in both the intended and the unintended effects of HEMI exposures is lacking. Additionally, a detailed understanding of human effects outside normal operating ranges and for novel applications is lacking. There is further a lack of understanding on how human effects to HEMI exposures vary among diverse populations and under special conditions (e.g., stress, heat). Military-relevant scenarios involve diverse populations and special condition, and novel HEMI weapons may be required to meet the unique requirements of military operations.

To support the DoD acquisitions process in meeting the military operational requirements, including the requirement for longer durations of incapacitation with longer standoff ranges, the HEMI Bioeffects Program of JIFCO enlisted the ARA team to develop a predictive modeling and simulation capability. To develop that capability, we have carefully considered the needs of both the warfighter and the acquisitions officer. That is, we have considered what information the acquisitions officer will be looking for when making funding decisions to support the warfighter. Toward that end, we began by answering the question, “Why do warfighters need HEMI devices?” This question was recently answered by the Commandant of the Marine Corps and Executive Agent for Non-Lethal Weapons, General Berger (Berger 2020).

Our potential adversaries will use any collateral damage – however lawful – to provoke escalation, generate negative views of America, or to erode political and public support for our operations. The unintended consequences of lethal force can undermine our legitimacy and may complicate our operations. Decisions to use – or forego – lethal force

¹ See (Reilly 2016) for an excellent survey of mathematical models of electrostimulation.

² e.g., Sim4Life <https://zmt.swiss/sim4life/>

can incur a heavy cost. Intermediate force capabilities enable U.S. and allied forces in delivering accurate, tailorable, and compelling effects in complex and ambiguous scenarios, while simultaneously preventing unnecessary loss of life or destruction of property. Our forces understand the strategic and political consequences of their actions; intermediate force capabilities can help mitigate those consequences.

We note that the operative words in the excerpt above are “accurate,” “tailorable,” and “compelling.” In this context, accurate means that the weapon elicits the desired effect, tailorable means that the desired effect may be modulated to meet operational needs, and compelling means that the desired first-order effect elicits an appropriate higher-order effect (e.g., disengagement) as needed. For HEMI weapons, as they are conventionally used in law enforcement, the desired effect is incapacitation of an individual for the purpose of physical control (Haskins, n.d.). By contrast, military applications could extend beyond control into active denial and deterrence to an individual and/or group of individuals and other higher-order responses.

In addition to the warfighter and the acquisition officer, we also considered HEMI device developers to provide broad applicability in the design of the HEMI M&S system. However, given the impetus of the program to support acquisition applications in support of military operations, the needs of the acquisition officer and of the warfighter took precedence in the design choices.

The HEMI M&S software system was designed specifically to build models of the effects of HEMI weapons with the unique needs of the military in mind. While making design choices, in the previous phase and in future phases of the project, we remember that the likely purpose of the HEMI weapon is to achieve some tactical objective that involves eliciting some desired behavior in individuals or groups without adversely affecting a strategic objective by causing significant injury to a person or people. Military operations are often about denial of capabilities. Nonlethal weapons provide warfighters with a means to achieve tactical objectives while denying the enemy the capability to exploit operational casualties for geopolitical gain. Officers will need to know, with precision and communicable certainty, what will be the outcome of an engagement. Planners will need to know if the objective will be achieved, and if not, what can be adjusted to meet the objective.

In the context of intermediate force capabilities, and HEMI weapons specifically, warfighters require an answer to two questions.

- 1) What is the probability that a HEMI weapon exposure will result in significant injury?
- 2) To what degree in the HEMI weapon a direct causal factor if someone is injured during a HEMI weapon exposure?

Note that there is no qualification of weapon type or exposure scenario in the first question. Warfighters need to know, particularly for acquisition applications, but also for other situations, what the risk of significant injury is for novel weapons in potentially unusual (i.e., not law enforcement) usage scenarios. And if injuries do occur, military operators need a defensible assessment of the causality.

The HEMI M&S that is being developed by ARA is designed specifically to answer those two questions. We are developing mechanistic (causal) models of the biological processes associated with HEMI exposure, and, more importantly, we are constructing extensible software

infrastructure to implement the models. In doing so, we are creating flexible modeling tools with durable utility to meet the ever-changing needs of the modern warfighter.

2. HEMI Mechanisms

HEMI weapons work by developing a voltage in the body that initiates bioelectromagnetic processes, i.e., those processes involving the direct interaction between biological entities and electromagnetic fields. The direct excitation of nerves is an example of a bioelectromagnetic process. The bioelectromagnetic processes can then initiate physiological processes that are causal descendants of the electromagnetic field, such as muscle contraction and reflexive nerve activation, which can activate additional processes and may, eventually, result in the outcomes of interest. Figure 2 shows the mechanisms of HEMI weapon effects at the highest-level. The common antecedent is electrodes contacting the body, and the outcomes are incapacitation (i.e., the target is subdued) and, potentially, the unintended effects (specifically, that the target is injured, but in general any unintended effect may be considered). In Figure 2, the mediating variables are hidden behind “the gap.”

The mathematical models that are described in the next section were developed specifically to represent the biological processes outlined here.

In this section, we expose the gap to describe the mechanisms, both known and hypothesized, that lead to the outcomes of interest, enabling the isolation and study of causal pathways. The mathematical models that are described in the next section were developed specifically to represent the processes outlined in this section. Figure 3 provides a first-level decomposition of the intermediate mechanisms of HEMI effects.

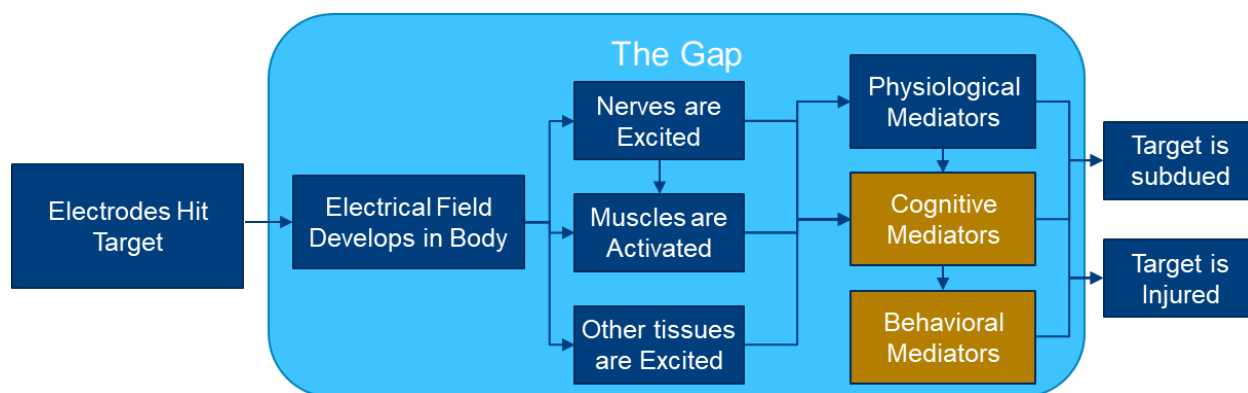


Figure 3. Mechanisms of HEMI exposure outcomes.

The gap in mechanisms is only partially expanded in Figure 3. There are multiple biological pathways by which the intended effects are elicited, thus the box in the figure labeled “Physiological Mediators” contains multiple pathways. Likewise, the box labeled “Target is Injured” contains multiple injuries. The modeling challenge is to mechanistically link the events of interest through biological processes. To do that, we begin with a conceptual model of HEMI mechanisms. Given the military objectives outlined in the previous section, the conceptual model begins with the development of an electromagnetic field in the body and the resultant bioelectromagnetic phenomena.

2.1. Bioelectromagnetic Phenomena

The bioelectromagnetic phenomena are processes that occur as a direct result of interactions between the electromagnetic field³ and biological entities. For HEMI weapon effects, the most important bioelectromagnetic process is the activation of excitable tissue due to development of an electric field. This process is described diagrammatically in Figure 4. After the electrodes contact the target and the device is activated, an electric field develops which causes a change in ionic currents and subsequent depolarization of the axon membrane. The propagation of this depolarization is called an action potential. The arrival of an action potential at the neuromuscular junction causes a subsequent action potential to develop in the membranes of the muscle cells of the innervated motor unit, leading to a contraction of muscle. The specific excitation-contraction mechanisms are discussed in the next section.

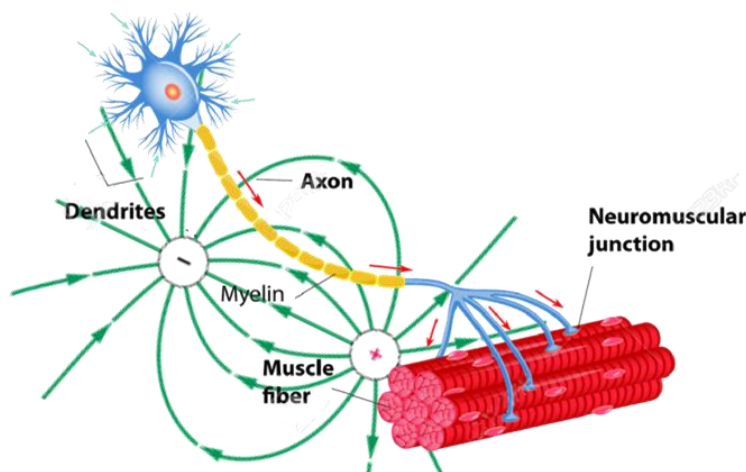


Figure 4. Example of a bioelectromagnetic process. An electromagnetic field causes a local ion current and subsequent action potential.

The bioelectromagnetic phenomena of conducted electrical weapons are well studied and generally understood. In fact, there is COTS software available for purchase for simulating the electromagnetic state of tissues given a stimulus (e.g., sim4life, ANSYS Maxwell). The threshold of stimulation of excitable tissues is also well studied, e.g., (Reilly 1988; 1998; Panescu et al. 2006; Ajeet 2008). Therefore, it is possible to predict, with a high degree of certainty, neurons in a body that will be excited by a given stimulus. For example, the volume of tissue containing excited neurons is sometimes called the “recruitment volume,” and in a previous phase of this project, we used an FDTD model of the body to compute current in tissues and then employed the SENN model (Reilly and Diamant 2011) to determine when the current met a threshold for nerve activation (Price, Asgharian, and Bellman 2018).

The loss of coordinated muscle control and resultant loss of posture that are typical of HEMI, and many of the injuries that sometimes occur following a HEMI weapon exposure, cannot be explained by bioelectromagnetic phenomena alone. The field strength generated by typical HEMI weapons is only strong enough to excite action potentials in excitable tissues that are in the anatomical region immediately surrounding the electrical contacts. Thus, the outcomes of

³ In the HEMI context, a field that originates or is powered ex vivo.

interest, i.e., incapacitation and significant injury, rely on physiological mechanisms that may be excited by, but which are distinct from, the bioelectromagnetic phenomena.

2.2. Physiological Phenomena

Physiology is a sub-discipline of biology that deals specifically with biological functions and mechanisms (the system of causally interacting parts and processes that produce one or more effects). In the context of HEMI effects, physiological phenomena are those which may or may not be causal descendants of bioelectromagnetic phenomena, but which do not rely on interaction with a non-biological electromagnetic field. In this subsection we describe the physiological mechanisms of HEMI effects.

2.2.1. Incapacitation (The Intended Effect)

The mechanisms that deny a target the ability to act deliberately are both bioelectromagnetic and physiological. The bioelectromagnetic mechanisms are primarily the direct excitation of muscle and the stimulation of motor neurons. Direct excitation of muscle occurs only in the area nearest the electrodes. Stimulation of motor neurons also occurs near the electrodes (at greater distances due to lower excitation thresholds), but the physiologic propagation of resultant action potentials can cause muscle to contract in anatomical regions far away from the electrodes.

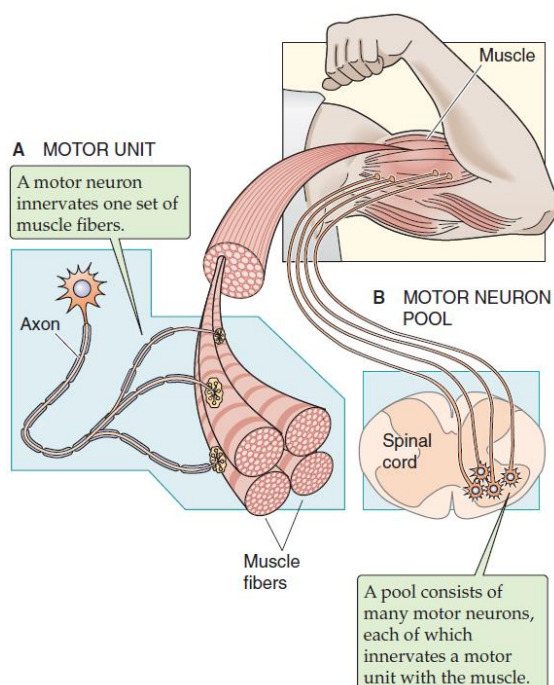


Figure 5. Motor units and motor neuron pools (Boron 2012).

Muscle innervation is shown in Figure 5. When a motor neuron is excited, an action potential travels along the axon membrane until it reaches the terminal end. One motor neuron can innervate multiple muscle fibers; thus, activation of a single motor neuron can result in contraction of multiple fibers. The neuron and the muscle fibers that it innervates are called a motor unit. The specialized synapse between motor neurons and skeletal muscle is called the neuromuscular junction.

When the action potential arrives at the terminal end of the axon (the nerve terminal), synaptic vesicles are released that contain a quantum of the neurotransmitter acetylcholine. Acetylcholine then crosses the synaptic cleft (the gap between the nerve terminal and the muscle fiber) and binds to receptors on the motor end plate of the muscle fiber membrane, causing a depolarization at the motor end plate known as an end plate potential (EPP) (Boron 2012). The process is summarized by Figure 6.

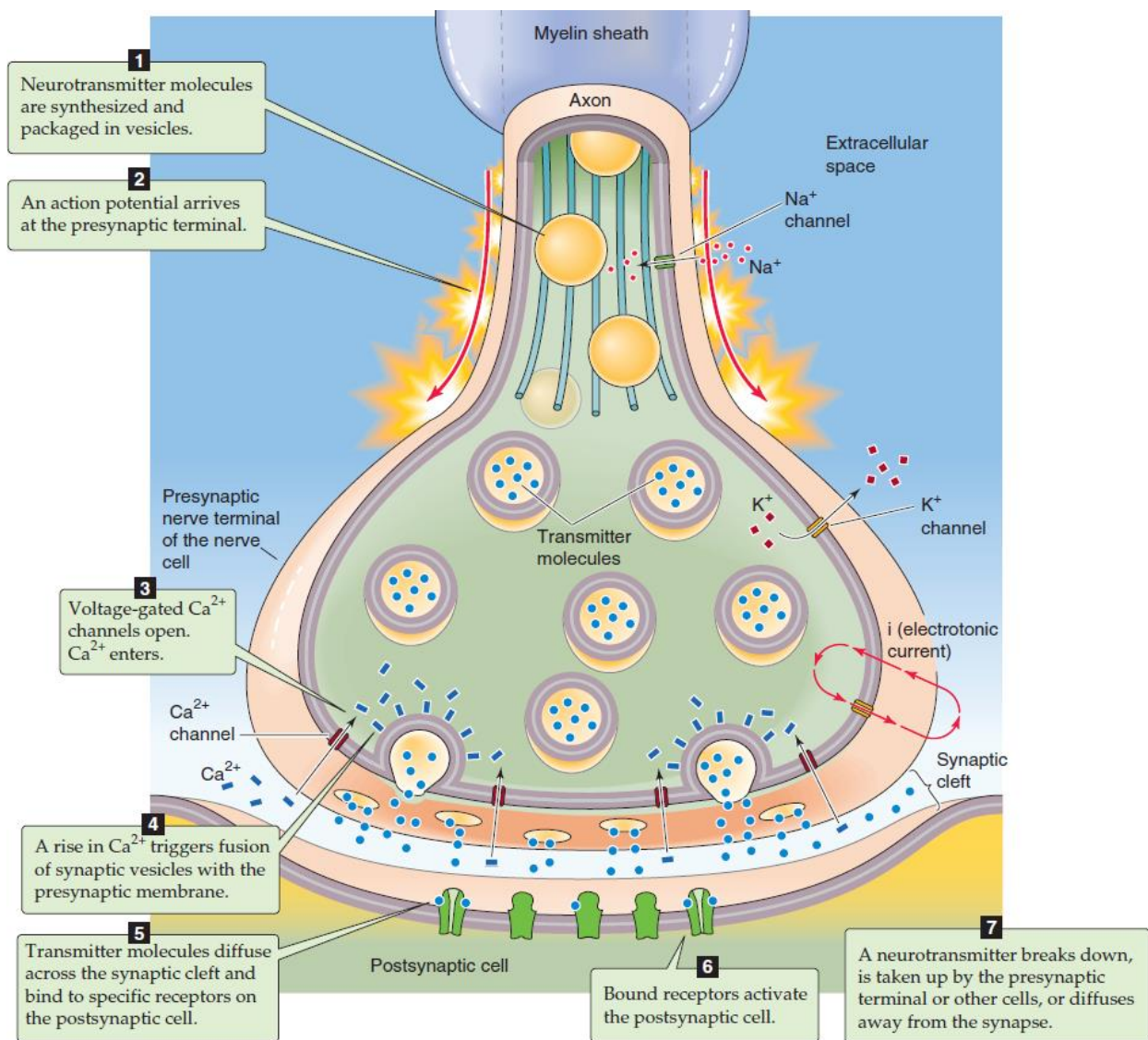


Figure 6. Synaptic transmission at the neuromuscular junction, reproduced from Figure 8-2 on page 215 in (Boron 2012).

Like an axon, muscle cells are excitable tissue, and when the EPP reaches a threshold value, and action potential spreads throughout the sarcolemma. This excitation triggers a process known as excitation-contraction coupling, shown in Figure 7, that is driven by the movement of calcium ions.

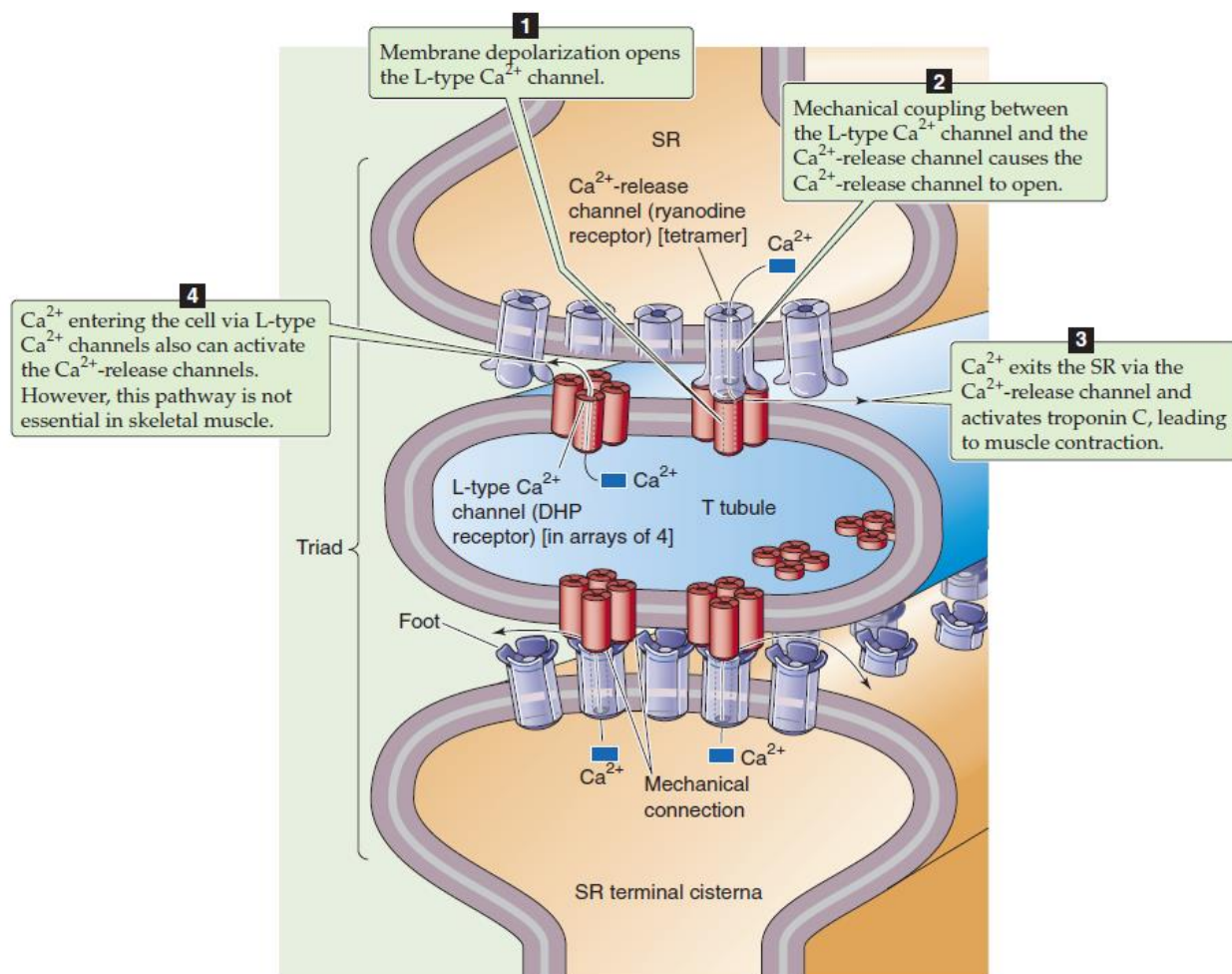


Figure 7. Excitation-contraction coupling in skeletal muscle, reproduced from Figure 9-3 on page 240 in (Boron 2012).

The causal pathway from excitation of the motor neuron through muscle contraction is common to both intended and unintended effects. However, nerve activation only tells a part of the story.

The primary physiological mechanism of the intended effect is likely a large-scale activation of reflex activity. In anesthetized pigs subjected to a HEMI weapon (Taser X26) and similar electrical stimuli, muscle contractions were observed in regions of the body far from the region where the current exceeds the threshold for activation (Despa et al. 2009). The remote contractions occur even with lidocaine blocking the nerve response at the stimulation site, indicating that reflex activity is the primary mechanism of widespread muscle activation.

2.2.2. Other Physiological Mediators (Unintended Effects)

The intended effect of a HEMI weapon is to subdue a subject without causing significant injury or death. However, unintended effects (e.g., injuries) do occur. In reviewing injury-related data in the TASER International database, we found that the potential for severe unintended effects of HEMI is low; however, the Risk of Significant Injury (RSI) associated with physiological effects such as renal damage, ventricular fibrillation, and seizures is not well-understood and difficult to quantify (Maier, Nance, Price, Sherry, and Reilly 2005; Maier, Nance, Price, Sherry, Reilly, et

al. 2005), particularly in a diverse population. Furthermore, the biological mechanisms of these effects are not well studied as empirical investigation requires expensive and time-consuming research using animal models. Likewise, research on HEMI use in a large, diverse demographic population has been limited, although some epidemiological data indicates that certain populations may be at greater risk than others. However, mechanistic models with explicit definitions of causality, whether known or hypothesized, can be employed to elucidate the relationships among HEMI device characteristics, physiological processes, and outcomes in diverse populations. See Section 7 for more information about the utility of explicit denotation of causal assumptions.

There are multiple injuries associated with HEMI weapon exposure, all with their own causal pathways. To date, our primary focus been on rhabdomyolysis and acute kidney injury.

2.2.2.1. Rhabdomyolysis and Acute Kidney Injury

Acute kidney injury (AKI), also known as acute renal injury, is a significant injury that has been associated with HEMI weapon exposure (Gleason 2015; Gross, Porterieko, and D’Andrea Joseph 2013) and other electrical injury (Brumback, Feeback, and Leech 1995), with rhabdomyolysis as the likely cause. Rhabdomyolysis is defined as a breakdown of muscle cells with release of intracellular contents into the blood, and it can lead to AKI through impairment of glomerular filtration (Bosch, Poch, and Grau 2009). The mechanisms of rhabdomyolysis include direct sarcolemmic injury and/or depletion of ATP leading to unregulated intracellular calcium. See (Bosch, Poch, and Grau 2009) for an excellent description of the mechanistic details of rhabdomyolysis and AKI.

Trauma leading to direct sarcolemmic injury is certainly possible following exposure to a HEMI weapon, especially in cases involving struggle or subsequent fall, but several factors (e.g. drug use and hyperthermia) often accompany HEMI exposure which may lead to non-traumatic cases of rhabdomyolysis (Gleason 2015; Sanford et al. 2011; Cooks et al. 2016b; 2016a; Gross, Porterieko, and D’Andrea Joseph 2013). Calcium dysregulation is the common factor in all non-traumatic categories of rhabdomyolysis. Prolonged HEMI exposure has the potential to initiate calcium dysregulation, particularly if electrically stimulated tetany is maintained long enough to deplete intracellular ATP stores and create a positive feedback loop of hypocalcemia-mediated tetany. The physiological process of calcium dysregulation and subsequent breakdown of the muscle membrane is discussed in the following subsection.

2.2.2.2. Calcium’s Role in Muscle Cell Damage

Calcium ions (Ca^{2+}) play an integral role in the functions of muscle. Ca^{2+} is used by all muscle types as their main regulatory and signaling molecule, with their function being controlled by Ca^{2+} as a second messenger (Berchtold, Brinkmeier, and Müntener 2000; H. Gissel 2005). Relevant for this work is the role of Ca^{2+} in the contraction and relaxation of skeletal muscle. It has been widely studied and described that Ca^{2+} is an essential component of the troponin-tropomyosin system for contraction and relaxation of skeletal muscle, with the binding of Ca^{2+} to troponin on the thin actin filament triggering contraction (Berchtold, Brinkmeier, and Müntener 2000). Due to its importance in muscle, the handling of Ca^{2+} within the muscle cell is vital for proper muscle function. Figure 8 below shows a visual representation of the key events in the skeletal muscle Ca^{2+} cycle. The figure shows that Ca^{2+} enters the myoplasm via the ryanodine

receptor (RyR) from the sarcoplasmic reticulum (SR), binds to troponin to initiate muscle contraction, is shuttled back to the SR by Ca^{2+} -ATPase, and binds to calsequestrin in the SR.

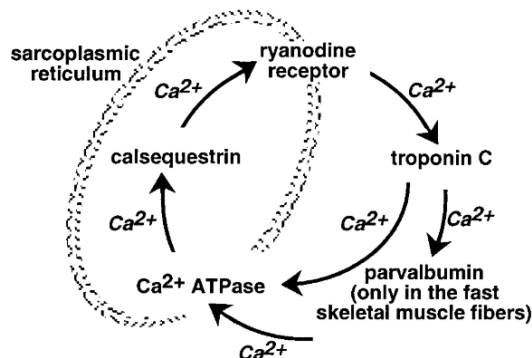


Figure 8. General overview of the calcium cycle in skeletal muscle, reprinted from (Berchtold, Brinkmeier, and Müntener 2000).

In the resting state of the muscle cell, the concentration of free cytosolic Ca^{2+} is maintained at about 50 nM (Berchtold, Brinkmeier, and Müntener 2000; H. Gissel 2005), while the extracellular fluid Ca^{2+} concentration is kept at about 1 mM (H. Gissel 2005). Therefore, there exists a continuous passive diffusion of Ca^{2+} into the muscle cells due to the enormous electrochemical gradient across the membrane (H. Gissel 2005).

To preserve low resting cytosolic Ca^{2+} concentrations, ATP driven pumps are needed to remove calcium from the cytosol (Berchtold, Brinkmeier, and Müntener 2000; H. Gissel 2005; MacLennan, Rice, and Green 1997; Bagur and Hajnóczky 2017). For example, the SR Ca^{2+} -ATPase, which has a high affinity for cytoplasmic Ca^{2+} , transports calcium back into the SR (H. Gissel 2005). The constant regulation of calcium levels in the muscle cell are import to prevent cell injury or cell death (Bagur and Hajnóczky 2017), however the ATP dependent nature of some of these regulation mechanisms may cause irregularities or dysregulation when ATP levels are depleted. Consequently, ATP kinetics and ATP depletion are included in our calcium dynamics model (see Section 3). Figure 9 provides an overview diagram of the relation between calcium and ATP dynamics.

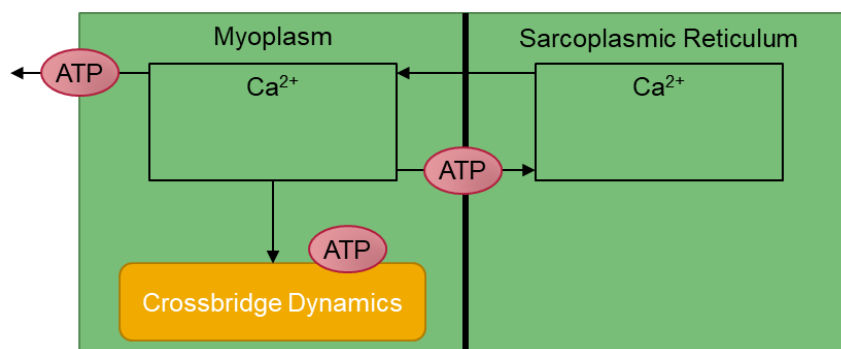


Figure 9. Conceptual overview of the relation between calcium and ATP dynamics.

2.2.2.2.1. Calcium Overload

The stimulation of the excitation-contraction-relaxation cycle is characterized by a large flux of Ca^{2+} from the SR to the contractile filaments and back (Hanne Gissel 2005; Fredsted et al. 2007), causing up to 100-fold increases in cytosolic Ca^{2+} concentrations (Hanne Gissel 2005). It has

been noted that prolonged increases in cytosolic Ca^{2+} concentrations at critical levels may lead to muscle cell damage (Fredsted et al. 2007; Hanne Gissel 2005). Several possible mechanisms for this calcium induced damage have been studied and described in the literature. A few examples are discussed in this section.

Calpains, Ca^{2+} -dependent cysteine proteases, are found in all vertebrate cells and are implicated in a variety of functions such as, sarcomere remodeling, signal transduction, cell cycle, apoptosis, and gene expression regulation (Huang and Zhu 2016). Because calpains take part in a variety of functions their overactivation/deregulation is implicated in several diseases (Huang and Zhu 2016) including diseases of the skeletal muscle (Vanderklish and Bahr 2000). Ca^{2+} is a major regulator of calpain activation, with increases in intracellular Ca^{2+} levels activating and translocating them to membranes (Huang and Zhu 2016). Substantial skeletal muscle proteolytic damage may occur due to this increased calcium concentration and calpain activation in the muscle cell (Huang and Zhu 2016). Sustained Ca^{2+} increases in skeletal muscle can produce damage such as, sarcomeric derangement, swollen mitochondria, swollen or fragmented SR, and lesions of the plasma membrane, these outcomes have been linked to calpain activation (A. N. Belcastro 1993; Angelo N. Belcastro, Shewchuk, and Raj 1998). Activated calpains can degrade membrane and cytoplasmic and nuclear substrates which can trigger apoptosis (Momeni 2011); another possible mechanism for induced muscle cell damage.

The calcium-dependent phospholipase A_2 (PLA_2), a membrane phospholipid cleaving enzyme, has been shown to be activated during contractions of skeletal muscle (Powers et al. 2011). Activated PLA_2 attack mitochondrial and other membrane phospholipids and stimulate the production of reactive oxygen species (ROS), all possibly contributing to muscle cell damage (Hanne Gissel 2005; Powers et al. 2011).

Mitochondria are major players in the process of muscle contraction-relaxation due to their role in ATP production and Ca^{2+} uptake (Zhou, Dhakal, and Yi 2016). Although Ca^{2+} uptake by mitochondria is essential for ATP synthesis, calcium overload in the mitochondria can set off a series of molecular malfunctions that may lead to mitochondrial damage or cell death (Zhou, Dhakal, and Yi 2016). Elevated cytosolic Ca^{2+} levels increase uptake of Ca^{2+} by mitochondria therefore, increasing the probability of opening the permeabilization transition (mPTP) pore, depolarizing the mitochondrial membrane potential, shutting down ATP production, causing massive Ca^{2+} efflux from mitochondria, initiating a detrimental feedback loop (Hanne Gissel 2005). This cycle can disrupt intracellular Ca^{2+} signaling and lead to muscle cell degeneration (Zhou, Dhakal, and Yi 2016).

2.2.2.2.2. Vicious Cycle Model

Several mechanisms and biological processes come into play when the contraction-relaxation phenomenon of skeletal muscle is induced by the action potential. Therefore, it is the interaction of these mechanisms and biological processes that lead to the adverse outcomes attributed to prolonged muscle contraction and sustained elevation of intracellular calcium levels rather than a direct relationship between an increase in cytosolic calcium concentration over an extended period of time and the destruction of the muscle cell membrane. However, the increase in cytosolic calcium levels and depletion of ATP are major drivers of a “vicious cycle”. The vicious cycle model is described diagrammatically by Figure 10.

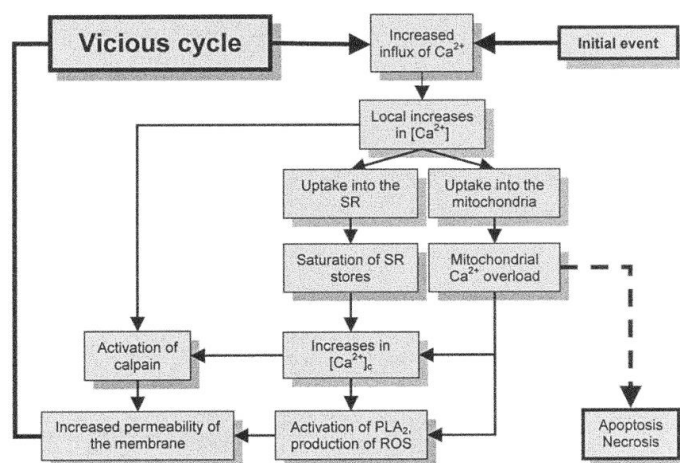


Figure 10. The “Vicious cycle” model of how increased influx of Ca^{2+} in the myoplasm can induce skeletal muscle damage, reprinted from Figure 2 of (Hanne Gissel 2005).

3. Mathematical Models

Any HEMI model built with the HEMI M&S System is necessarily an integration of several mathematical models. The composite model that is being built for the current effort includes one of two mathematical models of electrical dosimetry, a nerve model, and a muscle model. The current output of the muscle model, the intracellular calcium concentration, will be used to predict rupture of the sarcolemma and thereby the concentration of key cellular contents (e.g., creatine kinase, myoglobin) in the blood from which probability of rhabdomyolysis and subsequent renal injury may be computed. For brevity, we reference previously-described mathematical models provide detail only for the muscle physiology mathematical modeling in this report.

3.1. Bioelectromagnetic Models

The bioelectromagnetic models include the electrical dosimetry (also called “voltage”) models and the excitable tissue (nerve) models. Details about the electrical dosimetry mathematical model may be found in (Price, Asgharian, and Bellman 2018) and (Smith et al. 2019). The nerve model is also described in the aforementioned sources as well as in (Maxwell et al. 2020), and much more detail about the nerve mathematical modeling is available in (Reilly 1998) and (Reilly and Diamant 2011).

3.2. Renal Injury Model

A description of the mathematical model for predicting rhabdomyolysis and renal injury from plasma concentrations is provided in (Smith et al. 2019).

3.3. Muscle Physiology Model

When a nerve is excited by a HEMI weapon, an action potential travels along the nerve until it reaches the terminal end at the neuromuscular junction (see Section 2).

The muscle physiology model that we have developed for integration into the HEMI composite model describes the causal mechanisms between excitation of the nerve and both the intended effect (activation of skeletal muscle to override volitional control) and at least one unintended effect (rupture of the sarcolemma and subsequent rhabdomyolysis). The activation and propagation of the axon action potential is described by the nerve model. The muscle physiology model includes mathematical descriptions of the arrival of an action potential (AP) at the motor end plate (Section 3.3.1), the calcium dynamics of the excitation-contraction coupling (Section 3.3.2), and the force generation (Section 3.3.3).

3.3.1. Neuromuscular Junction

The mathematical model of synaptic transmission between a motor neuron and a muscle cells at the neuromuscular junction is described in this subsection.

As described in Section 2, when an action potential reaches the terminal end of a motor neuron, a biochemical signal propagates across the neuromuscular junction resulting in activation of an end plate potential and subsequent propagation of an action potential through the sarcolemma. In healthy human motor end plates, the amount of transmitter released per nerve impulse is greater than the amount required to trigger an action potential in the muscle fiber. The excessive biochemical transmission is known as the safety factor.

The safety factor, along with folds in the postsynaptic membrane that create a large surface area, guarantees a one-to-one relationship between the presynaptic and postsynaptic action potentials. Therefore, in our mathematical model, we assume that there is a postsynaptic action potential whenever there is a presynaptic action potential, obviating explicit modeling of the calcium dynamics and the release of acetylcholine at the motor end plate. Given the safety factor, we also assume that the frequency, amplitude, and wave forms of the presynaptic and postsynaptic membrane potentials are the same but that the postsynaptic potential is delayed due to the time required for the acetylcholine to cross the synaptic cleft. The relationship between the postsynaptic (end-plate) potential (V_m) and the presynaptic membrane potential (V_n) as a function of time and the delay is given by Equation (1).

$$V_m(t) = V_n(t - \tau_{syn}) \quad (1)$$

3.3.2. Calcium Dynamics of Excitation Contraction Coupling

As Figure 7 shows, calcium plays a critical role in excitation-contraction coupling. Likewise, calcium is thought to play a crucial causal role in the damage to muscle cell membranes that can lead to rhabdomyolysis (see Section 2.2.2.2). Therefore, we recognized early the importance of including the calcium dynamics in a model of muscle physiology for assessing the risk of acute kidney injury from HEMI weapon exposure. We had initially considered the detailed model by Shorten et al. (Shorten et al. 2007) for integration into the composite model, but the model would have required extension to account for the ATP depletion that is thought to cause calcium pump dysfunction with longer duration exposures (Metoyer 2017). Following a thorough review of the most recent literature, in which we were unable to find an existing model of excitation-contraction that would not require extension, we decided to extend a slightly less complex model (Otazu, Futami, and Hoshimiya 2001) and then implement the extension. We call the published lower-complexity model the Otazu model after the first author of the associated publication, and

we note that aspects of the Otazu model are incorporated into the more complex Shorten model. Furthermore, we note that both of the models fit within the generalized framework defined by the software architecture (see Section 5); therefore, use of the Otazu model in the current composite model should not be viewed as a decision to rule-out use of the Shorten model. On the contrary, it is likely that, in a future version of the software, we will include an extension of the Shorten model as an additional model of excitation -contraction coupling.

Following the foundation of the Otazu model, the relevant states of the muscle cell are the amount of calcium bound to Troponin, the amount bound to parvalbumin, the amount proximal to the sarcoplasmic reticulum, the amount distal to the sarcoplasmic reticulum, the fraction of activating sites bound to calcium, the fraction of inactivating site bound to calcium, the concentration of adenosine triphosphate (ATP) near the calcium pumps, and the muscle force. Note that the muscle force model is not the second-order model described by (Otazu, Futami, and Hoshimiya 2001) but rather the series component model described in (Wexler, Jun Ding, and Binder-Macleod 1997). Again, this decision was based on ease of integration and implementation, and it is important to remember that the software is specifically designed to facilitate integration of other models as user requirements evolve and knowledge of mechanisms matures. The force portion of the muscle model is described in the subsequent subsection.

A list of parameters and variables along with definitions and baseline values is included in this report as Appendix C. The excitation-contraction coupling model governing equations are shown below. The state equations are (2) through (8).

$$\frac{d[CaTN]}{dt} = \alpha_{trop}[Ca^{2+}]_{dist}([TN] - [CaTN]) - \beta_{trop}[CaTN] \quad (2)$$

$$\frac{d[CaPARV]}{dt} = \alpha_{parv}[Ca^{2+}]_{dist}([PARV] - [CaPARV]) - \beta_{parv}[CaPARV] \quad (3)$$

$$\frac{d[Ca^{2+}]_{prox}}{dt} = \gamma_{volt} + \gamma_{Ca} + \gamma_{leak} - \gamma_{pump} - \frac{[Ca^{2+}]_{prox} - [Ca^{2+}]_{dist}}{\tau_{prox}} \quad (4)$$

$$\frac{d[Ca^{2+}]_{dist}}{dt} = \frac{[Ca^{2+}]_{prox} - [Ca^{2+}]_{dist}}{\tau_{dist}} - \frac{d[CaTN]}{dt} - \frac{d[CaPARV]}{dt} \quad (5)$$

$$\frac{da}{dt} = \alpha_a(1 - a)([Ca^{2+}]_{prox})^2 - \beta_a a \quad (6)$$

$$\frac{di}{dt} = \alpha_i(1 - i)[Ca^{2+}]_{prox} - \beta_i i \quad (7)$$

$$\frac{d[ATP]}{dt} = -\frac{1}{2}\gamma_{pump} + \kappa(\beta - [ATP]) \quad (8)$$

The state equations are augmented by three rate equations and an equation describing the pump activation fraction.

$$\gamma_{Ca} = f_{Ca}(1 - i)a \quad (9)$$

$$\gamma_{pump} = \frac{f_{pump}[Ca^{2+}]_{prox}}{K_{pump} + [Ca^{2+}]_{prox}} g([ATP]) \quad (10)$$

$$g([ATP]) = \left(1 + \xi e^{-k_1([ATP] - \tau_{ATP})}\right)^{-\frac{1}{\xi}} \quad (11)$$

$$\gamma_{volt} = \begin{cases} 0 & \text{no pulse} \\ f_{volt} \sin(250\pi t) & \text{pulse start time} < t < \text{pulse end time} \end{cases} \quad (12)$$

3.3.2.1. ATP Depletion

As described in detail in Section 2.2.2.2.2, ATP depletion is thought to be a major contributor to the calcium dysregulation that can lead to breakdown of the sarcolemma after long periods of contraction. ATP is required by the ATPase calcium pump to move calcium against the electrochemical gradient after contraction. A reduction in ATP, or similarly, a diminished chemical potential of ATP due to the presence of ADP and P_i (Rockenfeller et al. 2020), is thought to contribute to dysfunction of ATPase resulting in positive feedback loop that leads to breakdown of the sarcolemma and subsequent rhabdomyolysis (Kim et al. 2016).

The dynamics of ATP production and consumption are complex. To provide energy for use in biological processes, ATP is dephosphorylated into ADP (and sometimes AMP). The ADP is then available for metabolic rephosphorylation. Thus, the total amount of ADP and ATP remains approximately constant. ATP is used by a multitude of energy requiring processes in the cell. To maintain model parsimony during development, we are currently only accounting for the ATP that is available to the calcium pump, and we are not modeling the mechanisms of rephosphorylation. Instead, we assume mass action dynamics for the rephosphorylation, as described by the second term in Equation (8). The dephosphorylation of ATP is modeled according to the stoichiometry of the pump, that is, one molecule of ATP is required to pump two molecules of calcium across the membrane, as described by the first term in Equation (8).

While it is known that the calcium pump requires ATP, the relationship between ATP depletion and the pump inhibition is complex (J. I. Bruce 2010; J. I. E. Bruce 2018). Rather than attempt to model the complex and not well understood mechanisms of pump inhibition, we instead chose a simple parametric inhibition model. We chose a generalized logistic model in the form of a Gompertz function, as shown in Equation (11), to define the ATP-dependent fraction of pump activity. Figure 11 is a plot of the calcium pump activity fraction as a function of the concentration of available ATP.

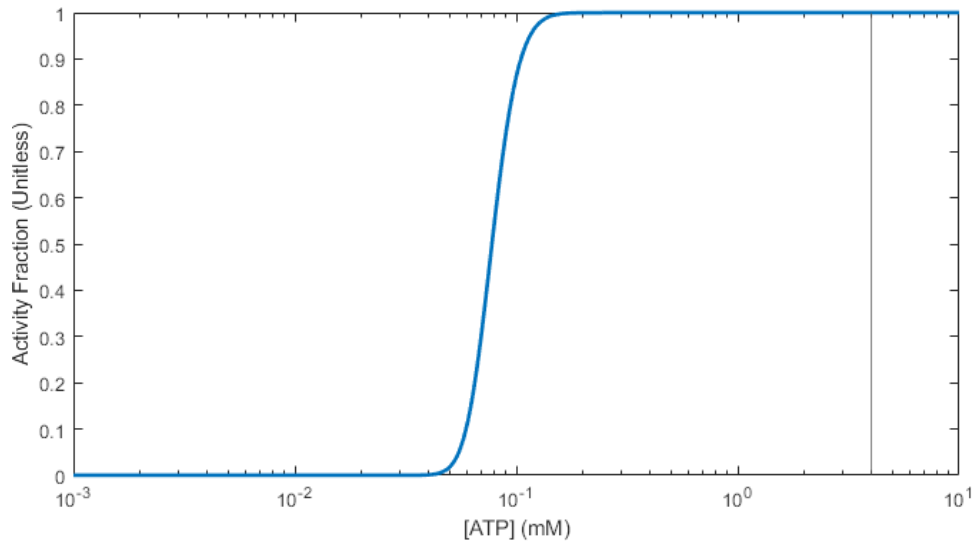


Figure 11. Calcium pump activity fraction vs. concentration of available ATP.

The resting concentration of ATP in muscle fibers is about 4 mM (Hall and Guyton 2011), and it is thought that the concentration of ATP must drop by orders of magnitude before pump function is inhibited (J. I. E. Bruce 2018). For example, the curve in Figure 11 shown that the pump activity fraction stays at 1.0 as the concentration falls well below the normal value, given by the vertical black line in the plot.

The calcium pump rate is also a function of the concentration of calcium proximal to the sarcoplasmic reticulum, as given by Equation (10). Figure 12 shows the calcium pump rate as a function of proximal calcium concentration for different values of ATP concentration, demonstrating the appropriate qualitative behavior of the model.

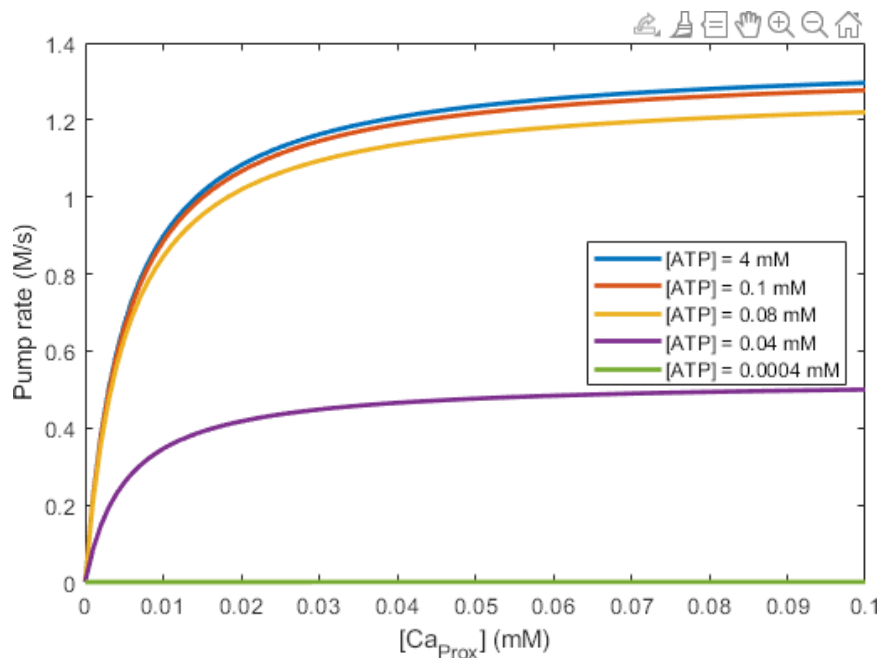


Figure 12. Calcium pump rate vs. proximal calcium concentration at different [ATP] values.

3.3.3. Muscle Force

We have implemented a muscle force model that is known to accurately predict isometric force of skeletal muscle in a murine model (Wexler, Jun Ding, and Binder-Macleod 1997), given by Equation (13). Note that the muscle force depends only on its own history and the current state of the activated troponin.

$$\frac{df}{dt} = A[CaTN] \left(1 - \frac{f}{F_m}\right) - \frac{f}{\left(\tau_1 + \tau_2 \frac{[CaTN]}{[TN]}\right)} \quad (13)$$

The force model is included as an example and is not currently being used for prediction.

4. HEMI M&S System Development Approach

There are many challenges associated with developing a modeling and simulation capability for HEMI weapons and with predictive model development in general. The solution to the key challenges is to build a flexible and extensible tool with which HEMI models may be constructed, modified, and adapted to meet myriad needs.

The HEMI M&S system development approach overcomes two key challenges and allows ARA and our partners to immediately develop useful software tools for military users even before requirements can be fully defined.

Challenge 1) At this stage of the program, the user requirements and use cases cannot be perfectly defined.

The program goal is to develop a predictive modeling and simulation capability to support the DoD acquisitions process in meeting the military operational requirements, including the requirement for longer durations of incapacitation with longer standoff ranges. To meet this goal, we know that we need software that executes models to make predictions of the efficacy of CEWs and of the risk of significant injury for novel HEMI weapons and exposure scenarios. However, the character of the novel weapons and the nature of the exposure scenarios will likely change to meet different military objectives. Acquisition officers will likely have a broad range of requirements for the HEMI M&S tool.

Challenge 2) Knowledge of the biological mechanisms of HEMI outcomes is evolving and will continue to evolve.

As described in Section 2, the mechanisms of HEMI-related outcomes are complex, and many are a subject of ongoing research. As knowledge of the biological function advances, so too will the models that describe that physiology.

Without perfect knowledge of the user requirements at the start of development, the choices are to 1) build a model that meets one potential requirement well and ignores the others, 2) create a system that meets all possible requirements, 3) build an extensible solution. The first option is risky as the chosen requirement may not fit well with the eventual acquisition application. The second option is ideal but

Our solution is to build a flexible and extensible modeling and simulation system.

impossible to achieve given finite resources. Option three is the only feasible approach to achieve the goals of the HEMI Bioeffects Program.

4.1. Common Data Model

As we develop the HEMI M&S system, the primary objective is to build infrastructure that supports further development and extension. Toward that end, we have developed a modular and extensible software architecture, shown in Figure 13. The architecture is described in detail in Section 5.

The core of the software architecture is a set of standardized, extensible data schemas called a common data model (CDM). The CDM is a specification of all the data and relationships associated with HEMI models and it provides a well-defined data interchange format that disparate models can use for standardizing information exchange. The CDM, in conjunction with the rest of the software architecture described by Figure 13, allows developers from any organization to modify existing models and add new models while maintaining a base level of interoperability.

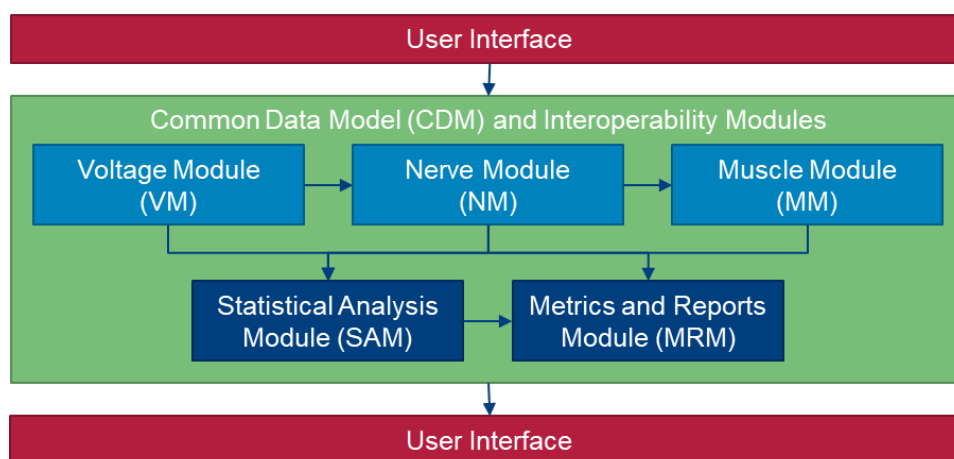


Figure 13. High level operational viewpoint of the HEMI M&S system software architecture. SAM and MRM are future modules.

The traditional way to ensure that disparate models can communicate with one another is to develop one-to-one interfaces between them, establishing syntactic interoperability between the models. When these composite models are developed by a single entity (e.g., an individual developer or development team), semantic interoperability is often established as well, albeit implicitly. This traditional approach can be effective and efficient, particularly for development of computational models with a well-defined purpose that will not require future modification or extension of either the constituent models or the modeling and simulation system.

In cases where the modeling and simulation system will likely require future extension and/or modification, e.g., when the system goals are broad, the traditional architectural design presents challenges. The addition of new models into the system or improvements to existing models requires an update to the various interfaces which entails writing new and/or refactored code. As the complexity of the simulation system increases, so does the cost of implementation of any change. Even just maintaining the simulation system throughout its lifecycle can require significant costs if the system has any external dependencies.

The CDM is a data-centric approach that provides a more natural way of abstracting and implementing the solution when the purpose of a simulation system is broad or likely to evolve with an increased understanding of the domain.

4.2. Configuration Management

We are developing the software using git for version control and GitLab for managing development operations (DevOps). This allows us to maintain a complete history of the software development process (see figure below) to facilitate future extension and to enable collaboration with other persons and organizations.



Figure 14. Screenshot of a git history graph for the HEMI M&S System.

We did not plan to develop a continuous integration and continuous delivery (CICD) pipeline during this phase of the project, and thus we are not spending resources on automated build, test, and deployment tools. However, we recognize that these tools are important during the lifecycle of a credible M&S tool. Therefore, since we are using GitLab for version control, we are mindful of the CICD deployment tools available on the GitLab platform, and we are designing the software such that the future integration of these tools to establish CICD a pipeline is fully supported.

4.3. Development Philosophy

Simply stated, our development philosophy is to maximize the potential for utility. Our approach allows us to move forward with model and software development in the face of imperfect knowledge. We start with the assumption that we are going to be wrong about some things, and we architect the software to allow us and/or others to adjust the core functionality as knowledge improves and needs transform.

5. Software Architecture

A high-level description of the software architecture is provided by Figure 13. The modules shown in Figure 13 are parent classes from which model classes are derived, and the CDM provides the foundation for integration. This section of the report provides a detailed description of three of the HEMI M&S system modules and the models that are implemented using the module framework and integrated through the CDM. The nerve, voltage, and muscle modules and implemented models are described below. The statistical analysis and metrics and reports modules are future modules that are currently in the design stage. These modules will be described in detail in future reports and are discussed conceptually in Section 8.1.1.

5.1. Nerve Module

This subsection begins with an overview of the HEMI M&S system nerve module followed by a description of the only nerve model that has been implemented in the HEMI M&S. Note that the modular architecture enables implementation of multiple nerve models, though, as of this writing, there are no plans to implement additional nerve models.

5.1.1. Overview

The base class for nerve modules is defined in `SENerve.h` and is called `SENerve`. All nerve modules to be developed for the HEMI software should inherit from this base class. It is assumed that all nerve modules developed to work with the HEMI Modeling and Simulation software will have the following in common:

- For a given model, more than one nerve may be instantiated so that nerves must be distinguishable from one another.
- They will all need to specify coordinates for the locations of a neuron's nodes of Ranvier (referred to below as "nodes").
- They will all require a means of specifying the values of external voltages at the locations of the nodes.
- They will all use the values of external voltages so obtained, along with zero or more values of other input parameters which may be required to calculate the values of transmembrane potentials at the locations of the nodes as a function of time.
- They may need to provide the transmembrane potentials calculated to other modules. More specifically, they may need to provide the transmembrane potentials calculated for the distal or proximal end of the nerve.

With these needs in mind, the public interface for nerve modules is defined in `SENerve` as follows:

- A protected data member `m_name` of type `std::string` is specified to distinguish internally one instantiation of a nerve from another. Public methods associated with this name include:
 - `HasName` which returns `true` if a nerve object's name has been set and `false` otherwise.

- `SetName` which takes a `std::string` as an argument and sets `m_name` to its value.
 - `GetName` which returns a `std::string` whose value is the value of `m_name`.
 - `InvalidateName` which renders `m_name` empty.
- A protected data member `m_coordinates` which is defined to be a `std::vector` container with elements that are pointers to the type `SEPointType`. This latter type is a typedef defined in `CommonDataModel.h` to be a `std::tuple` whose elements are of type pointer to `SEScalarLength`. `SEScalarLength` in turn is part of the Common Data Model and is a container for parameters with units of length. A pure virtual function called `SetNodeCoordinates` is also declared, and takes three `std::vector` arguments of type `double` for holding the x, y, and z coordinates of nodes respectively and an additional argument of type `LengthUnit` which specifies the units to be assumed for the coordinates contained in the vectors. This function must be overridden by all classes deriving from `SENerve`.
- A protected member `m_voltage` of type pointer to `SEVoltage`. This member variable is to be assigned to point to whatever object serves as a module for voltage for the simulation and is required to access that module's public interface. A public, pure virtual member method is also provided so that `m_voltage` can be set to a particular voltage module object. This function must be implemented by developers of nerve modules which derive from `SENerve`.
- A protected member `m_external_potentials` which is defined to be a `std::map` whose keys are of type pointer to `SEPointType` (see above), and whose values are of type pointer to `SEScalarElectricPotential`. This latter type is part of the Common Data Model and is a class designed to contain data values intended to represent electric potential. Associated with this member is a public pure virtual function which takes a map of the same type as `m_external_potentials`, as its argument. This function must be implemented by derived classes but is intended for the purpose of filling `m_external_potentials`.
- A pure virtual function called `ConfigureFromFile` which takes two arguments of type `std::string`. The first argument is a string providing the name of a parameter file containing names, values, and units for input parameters presumed to be required by the module implementing the function. The second argument is a string providing the path to the directory where the parameter file is supposed to be located. A data member `m_working_directory` can be set to the value of this path if a module needs a location to output data files. Related to `ConfigureFromFile` are the four pure virtual serialization methods called `SerializeToString`, `SerializeToFile`, `SerializeFromString`, and `SerializeFromFile` which must be implemented for each file deriving from `SENerve`. These serialization methods can be used in implementations of `ConfigureFromFile` to read in parameter data from JSON

formatted input files. They can also be used to generate output files of the same format to be used later as input files as desired.

- A protected member called `m_membrane_potentials` defined as a `std::vector` containing elements of type pointer to `SEScalarElectricPotential` and intended to contain values of transmembrane potentials evaluated by the nerve module based upon the model parameters and the values contained in `m_external_potentials`. Exactly how the elements of `m_membrane_potentials` are calculated is left to the developer of nerve modules deriving from `SENerve`, though it is assumed that knowledge of a current time, and a current time increment will be needed and that some kind of time stepping will be required. For these latter needs the interface includes pure virtual public methods `SetCurrentTime` (taking an argument of type pointer to `SEScalarTime`), `SetCurrentTimeIncrement` (also taking a pointer to `SEScalarTime`), and `TimeStep` (taking a pointer to a voltage object and a pointer to `SEScalarTime`). These must all be implemented by the developer of a nerve module. Examples of their implementation can be seen in the one nerve module currently provided as part of the HEMI software called `SEsennNerve` to be described in section 5.1.2
- The developer of a nerve model must also implement the following public methods specified as pure virtual functions in the base class:
 - `GetMembranePotentials` for providing a reference to a `std::vector` with elements of type pointer to `SEScalarElectricPotential` containing the calculated values of the transmembrane potentials.
 - `SetDistalNode` taking an unsigned `int` as a reference to be used to identify which node is to be identified as the distal node of the nerve. The integer specifies an index which can be used as subscripts for the `std::vectors` `m_coordinates` and `m_membrane_potentials`.
 - `SetProximalNode` also taking an unsigned `int` to be used to specify an index to `m_coordinates` or `m_membrane_potentials` and identifying which node is the proximal node of the nerve.
 - `GetDistalNodeVoltage` and `GetProximalNodeVoltage` for obtaining and returning the values of the membrane potential at the distal node and proximal node respectively. These methods are provided for use by other modules (particularly the muscle module).
- Finally, all modules, whether, nerve, voltage, or muscle, must implement an `Initialize` method. For some modules, this method may be empty, but in anticipation of the possibility that the other public methods provided for module interfaces such as `ConfigureFromFile` are insufficient for fully initializing a module prior to the commencement of time stepping, the pure virtual method `Initialize` is defined in `SENerve` base class. The `Initialize` methods of all modules are called by the `Initialize` method of the `Controller` class prior to execution of a simulation.

5.1.2. SENN Nerve Model

Currently the HEMI Modeling and Simulation Software has only one option for nerve modules, namely the SENN Nerve Module defined by the class `SEsennNerve` which inherits `SENerve` and whose definition can be found in `SENerve.h` (with method implementations found in `SENerve.cpp`). As suggested by the name, `SEsennNerve` provides the same model for nerves as is provided by the original SENN code described by (Reilly and Diamant 2011).

Protected data members are defined for all parameters required for a SENN nerve model using types available in the Common Data Model. Those that were not user-settable in the original SENN code are initialized in the constructor for `SEsennNerve` (though these settings can be overridden in HEMI by resetting their values in the JSON formatted input file that can be read by `SEsennNerve`'s implementation of `ConfigureFromFile` or by hand-setting them using provided setter methods).

Those data members not initialized in the constructor must be set by the user via a call to `ConfigureFromFile` which has been implemented to allow for backward compatibility with the SENN code's format for its input file `inparam.txt` (although `inparam.txt` must be modified to include the pulse start time parameter (PST) and the parameter RHOE in the `STIMULUS NAMELIST`). If the user specifies a SENN style input file, `SEsennNerve`'s implementation of `ConfigureFromFile` reads from this file in the same way `ObjSENN` reads from it. Otherwise, if a JSON formatted file is provided (which can be generated by a call to `SerializeToFile` when running a HEMI simulation that reads data in from a SENN style input file) `ConfigureFromFile` will read the JSON formatted input. (This is the preferred method of input for the HEMI Modeling and Simulation software.) Note that although setter methods have been provided as part of the public interface of `SEsennNerve`, it is not currently recommended that these be used as an alternative to `ConfigureFromFile` as the latter file is currently implemented to also set some parameter values which depend on the values input for their value. The setter methods are included to facilitate serialization.

The rest of the `SEsennNerve` module is essentially a reimplement of portions of the SENN code using `ObjSENN` as a model for the implementation, and with specification of external potentials left as a task for a voltage module to handle (see section 5.2). Note that if a voltage module of type `SEsennNativeVoltage` is chosen for a simulation, it is possible to recover results for transmembrane potentials that match the results of `ObjSENN` itself. The portion of `ObjSENN` which is reimplemented in `SEsennNerve` is that portion responsible for carrying out a single timestep for integration of the SENN mathematical model for a nerve. This model is described in detail in [need to cite Reilly again], and so we will not reproduce it here. We limit ourselves to stating that this model consists of a set of ordinary coupled non-linear differential equations of the form of Equation (14), where, if n_R is the number of nodes of Ranvier and n_{nl} is the number of those nodes that are nonlinear, then $N = n_R + 4n_{nl}$.

$$\frac{dy_i}{dt} = f(y_1, \dots, y_N; t) \quad (14)$$

Internally, `SEsennNerve` represents the vector of solutions to this set of equations with the standard vector of doubles called $\mathbf{y}$, while values computed for the right-hand side of these

equations are contained in the standard vector of doubles called `dydt`. These are both private members with the first n_R members of `y` containing the current values of the computed transmembrane potentials themselves. The vectors `y` and `dydt` are initialized prior to the commencement of time stepping in a HEMI simulation via a call to `SEsennNerve`'s implementation of the `Initialize` method.

Like `ObjSENN`, `SEsennNerve` makes use of a library function from the Boost library `odeint` to carry out a single step of integration of the SENN set of differential equations. Currently the `TimeStep` method of `SEsennNerve` is written to use Boost's implementation of the Dormand-Prince Runge-Kutta method of orders 4 and 5 (called `runge_kutta_dopri5` and instantiated as `rk` in `TimeStep`). In addition to the vectors `y` and `dydt`, the current time, and the time increment, this implementation's stepper—called `do_step`—requires that a function or function object be passed for the purpose of evaluating the values `dydt` at each partial timestep required by the solver. To serve this purpose, a class internal to `SEsennNerve` called `Senn` (and based on its namesake found in `ObjSENN`) is defined and instantiated as part of the definition of `SEsennNerve`. The actual instantiation of `Senn` is called `sn`. Within the definition of `Senn` the parentheses operator is overloaded so that `Senn` is understood to be a function object which can be passed to `rk.do_step` as required by that method in `TimeStep`.

The action of the overloaded parenthesis operator of `Senn` (that is, `Senn::operator()`) is essentially that of the original SENN code's subroutine called `FCT`. It begins by determining the values of the external potentials at the nodes of the nerve. In the present case, this is carried out by a call to the voltage module's public interface method `EvaluateExternalPotentials`. (This method is available via the pointer member `m_voltage` mentioned above which (by the time `Senn::operator()` is called for the first time in a simulation) should point to the instantiated voltage module. After obtaining these, it then calculates those portions of the right-hand sides of the SENN model differential equations which involve these external potentials and the current values of the transmembrane potentials held in the vector `y`. Then it completes the calculation of the right-hand side (`dydt`) assuming that all nodes are of linear type.

The last portion of `Senn::operator()` involves a loop over the nonlinear nodes only⁴ and reevaluates the corresponding values of `dydt` to account for the nonlinear current densities associated with the nonlinear nodes. This reevaluation is done—as in the original SENN code itself—on the basis of the Frankenhauser-Huxley model for the electrodynamics of a myelinated nerve fiber (Reilly and Diamant 2011).

5.2. Voltage Module

The purpose of the Voltage Module, regardless of the form of the model, is to provide as input to the excitable tissues models the bioelectromagnetic state of the surrounding tissues. First, an overview of the voltage module is provided, followed by a description of the two voltage models that are currently implemented in the HEMI M&S system.

⁴A relic of the original SENN code that allows some nodes to be specified as nonlinear while the rest are linear. This option is retained in both `ObjSENN` and HEMI, but in practice all nodes are typically taken to be nonlinear.

5.2.1. Overview

Voltage modules are understood to take as input, a set of coordinates specifying locations within the volume of the stimulated subject where the value of the electrical potential induced by the stimulation is required to be known at some given time. In particular the voltage module is intended to provide the values of the “external potential” at the locations of the nodes of Ranvier of a neuron and at a specified time. These potentials represent the values of the electrical potentials at the locations of the nodes and external to the cell membrane and are required as input to nerve models for calculating voltages across the cell membrane (the transmembrane voltages) at these locations and the given time. The locations of the nodes are part of the specification of the neuron and must be passed to a voltage module from the nerve module’s function object for the calculation of the right-hand-side of the nerve model’s set of differential equations.

The public interface required for voltage modules is defined in the base class `SEVoltage` found in `SEVoltage.h`. The base class specifies just two protected data members, namely the string `m_working_directory` for specifying locations where files may be found and written to, and the map `m_electric_potentials` which serves as the container for the external potentials to be provided by the voltage class to the nerve class accessing its interface. This map is a C++ standard ordered map whose keys are the coordinates where the external potential values are required and whose values are the external potentials themselves. The map’s keys are of type pointer to `SEPointType` which is a typedef declared in `CommonDataModel.h`. It is a C++ standard tuple whose three elements are pointers to `SEScalarLength` as defined in the CDM. The map’s values are of type pointer to `SEScalarElectricPotential` also as defined in the CDM.

Aside from these data members the base class specifies several pure virtual member methods to be overridden by methods implemented for a specific voltage class which derives from the base class. Chief among these is the method `EvaluateExternalPotentials` which takes a reference to a standard vector of pointers of type `SEPointType`, and a pointer to an object of type `SEScalarTime`. The former of these provides the locations where external potentials are to be specified and the latter provides the time for which they require specification. `EvaluateExternalPotentials` is expected to fill the `m_electric_potentials` container and return a reference to it to the calling method of the nerve object.

The base class also specifies the following pure virtual methods:

- `SetTimeIncrement`, in anticipation of the need for a voltage module to require knowledge of the time increment for integration, and which takes as its argument a pointer to an object of type `SEScalarTime`.
- `ConfigureFromFile`, in anticipation of the need for a voltage module to read values of parameters for data members specific to the module from a file. This method expects one string providing the name of the file to be used for configuration and another string providing a path to the location of the file specified by the first string.
- `SetRecordVext`, in anticipation of the desire to create a time series record of the external potentials calculated by the module.

- `Initialize`, in anticipation of the need to finalize the configuration of the voltage module prior to its use in simulation time marching.

Finally, classes derived from `SEVoltage` must also override a set of pure virtual serialization methods included in anticipation of the specific input/output requirements of a voltage class so derived. These are expected to require the handling of differing parameter sets for differing derived voltage classes and hence are the responsibility of the voltage module developer. Implementations of the overrides of these methods found in voltage classes already provided with the HEMI software will likely serve to illustrate how such methods should be implemented by developers of additional voltage models. These serialization methods are common to all HEMI module base classes whether of the nerve, voltage, or muscle type and are a part of the CDM infrastructure (see Section 4.1).

The HEMI Modeling and Simulation Software currently provides two separate voltage model classes derived from `SEVoltage`. These are `SESennNativeVoltage`, and `SEFDTDVoltage`. We provide descriptions of each of these in the following two subsections.

5.2.2. SENN Native Voltage Models

A goal of the HEMI Modeling and Simulation software is to provide voltage models for providing external potential values which are obtained from the most realistic models of electrical dosimetry possible. Hence our inclusion as a proof of concept of the FDTD voltage module. However, for testing purposes, and perhaps other purposes as well, it is useful to include a simplified model for external potentials based on previous work. In particular, because the only currently implemented nerve module in the HEMI software is one taken from the SENN model, it is only reasonable to include a voltage module in the HEMI software which provides at least some of the same modeling options for external potentials as the original SENN model itself provides. By coupling the two modules together in a HEMI simulation it should thus be possible to reproduce the results for transmembrane potentials that one would obtain from the SENN code itself. As part of the development of the HEMI software we have implemented such a module which we call `SESennNativeVoltage`.

In an earlier phase of the HEMI project, we developed an object-oriented C++ language reimplementations of the original SENN code for nerve stimulation modeling called `ObjSENN`. Most of the logic and nearly all of the functionality of the SENN code is preserved in `ObjSENN`, and a strong resemblance between the inputs and outputs of `ObjSENN` and those of the original SENN code was intentionally maintained. As noted elsewhere, many tests were completed on `ObjSENN` to verify that its results match those of the original SENN software. Most of the implementation of `SESennNativeVoltage` is borrowed directly from `ObjSENN` with only minor modifications required to account for differences in naming conventions and class ownerships. The underlying decision trees and mathematical models employed by `ObjSENN` (and hence SENN itself) are precisely identical to what appears in these earlier work products. Because both `ObjSENN` and SENN are well-documented elsewhere we content ourselves here with a somewhat generalized overview of the implementation of `SESennNativeVoltage`.

5.2.2.1. `SEsennNativeVoltage::EvaluateExternalPotentials`

We recollect from our discussion of the base class for voltage modules that the principle public member method for voltage modules is called `EvaluateExternalPotentials`. In this section we discuss its implementation for the SENN Native Voltage module.

In the original SENN code, the external potential values for a given timestep are calculated during roughly the first half of the execution of a call to the subroutine `FCT` found in that code. That section of code was translated into C++ and, in `ObjSENN`, can be found in a method of `ObjSENN`'s `Senn` class called `calculate_ept`. In `HEMI`, the same logic can be found in `EvaluateExternalPotentials` (with some additional code required to obtain internal representations of required parameters from the Common Data Model). While appearing very different superficially, the logic in all three implementations is essentially the same. We outline it as follows referring in particular to the implementation found in `HEMI`:

- The first step is a call to the private member method `set_epot` which takes a vector of coordinates of “nodes” where values of external potentials are needed. This method calculates an amplitude for the external potential at each node, determined from a current amplitude or current density amplitude selected by the user, and from the locations of the nodes relative to the locations of the electrodes of the device providing electric current. The external potential amplitude is emplaced as an element of a vector called `m_epot`. Details regarding how `m_epot` is evaluated are dependent upon how options for electrode type and electrode environment are set and are described in (Reilly and Diamant 2011). These options correspond to the possible values for the parameters `FS` and `S` from the original SENN code. Options for both point (`FS=0`) and wire (`FS=3`) electrodes are maintained in the `HEMI` implementation as well as both surface (`S=1`) and *in situ* (`S=0`) options for electrode environments. However, specifications of `m_epot` via reading from a file (`FS=2`), and specification of a uniform field parallel to a nerve axis (`FS=1`) are not implemented in `HEMI`.
- The values in `m_epot` provide information regarding how the external potentials vary spatially depending on node location but must still be scaled according the temporal waveform chosen. The next step carried out by `EvaluateExternalPotentials` is the determination of the scaling factor required. If the time for which `EvaluateExternalPotentials` falls within a range of times during which the stimulus is applied, the values calculated for the external potentials are determined by multiplying this scaling factor by the values in `m_epot`.
- The scaling factor (called `xmult`) depends firstly on the value chosen for the `TemporalWaveFormFlag` specified by the user in an input file. This flag corresponds to the original SENN code's parameter called `IWAVE`. In `EvaluateExternalPotentials` this quantity is referred to locally as `iwave`. As in SENN, the choice of `iwave` determines the choice of temporal functional form for the electrical stimulus. For example, `iwave=1`, codes for a monophasic pulse or pulse train. For this case, the scaling factor is set to unity and the only time-dependence entering into the determination of the values of external potential involves determining whether the

pulse is “on” or not at the time for which the potentials are requested. (If the time exceeds the pulse duration or falls within a period between multiple pulses of a pulse train the external potentials are set to zero.)

- As another, somewhat less trivial example, if `iwave=2`, the temporal waveform is sinusoidal. In this case, a frequency f and phase shift φ must be specified by the user. Then, if the time t lies within the range of duration for the pulse, the scaling factor is set to $\text{xmult} = \sin(2\pi ft + \varphi)$. This value then multiplies the values in `m_epot` to determine the values for the external potentials provided as the result of the call to `EvaluateExternalPotentials`.
- `SEsennNativeVoltage` implements some, but not all of the options for `iwave` available in `SENN` and `ObjSENN`. Others included (besides `iwave=1` and `iwave = 2`) are as follows:
 - `iwave = 4` (exponential pulse with adjustable time constant τ_s .)
 - `iwave = 5` (sinusoid with exponential envelope.)
 - `iwave = 6` (biphasic pulse doublet with differing amplitudes for each phase.)
 - `iwave = 10` (Additive pair of sinusoids).
 - `iwave = 11` (cosine amplitude modulation of a sinusoid).
 - `iwave = 13` (external current data read in from file).

5.2.3. FDTD Voltage Model

Here we describe the high-resolution, high-complexity FDTD-based voltage model. We begin with an overview that includes discussion of the need for time averaging due to a transiently non-conservative field.

5.2.3.1. Overview

The FDTD voltage module—called `SEFDTDVoltage`—is designed to work with time-averaged data generated from the output of the OpenEMS-based Electrical Dosimetry module. The Electrical Dosimetry module is not currently integrated with the HEMI Modeling and Simulation software, but rather exists as a separate entity in the form of a set of MATLAB scripts which interface with the OpenEMS electromagnetic field solver. OpenEMS uses the Finite Difference Time Domain method (FDTD) to solve for the electromagnetic field as a function of time in a volume with geometry and electrical properties specified by the user. As part of our software package, we provide MATLAB scripts that use OpenEMS to calculate a time series of the components of the electric field at certain mesh points within a volume presumed to represent the body of a HEMI weapon test subject. In particular, for purposes of development and testing during this phase we have been using the script called `electricalDosimetry_v6_pig.m` which models HEMI exposure of a porcine model.

The output generated by the MATLAB scripts such as `electricalDosimetry_v6_pig.m` is provided in the form of an HDF5 file called `Et1h5.h5`. This file contains two important groupings of data. The first of these—called “Mesh”—consists of three datasets which together provide the Cartesian coordinates of the mesh points in units of meters at which values of the components of the electric and magnetic fields are calculated within the volume of the test

subject. These mesh points remain the same for all times at which values of the field components are recorded. The second grouping of data, “FieldData/TD” contains datasets of the components of the electric field, with each dataset output for a specific time which is provided as an attribute to the dataset. The names of these datasets correspond to the OpenEMS timestep number for which the data is output. For example, if the oversampling rate has been selected so that a dataset is output once for every 28 OpenEMS timesteps, the first three dataset names will be “00000028”, “00000056”, and “00000084.”

As implied above, the frequency with which field data is output to `Et1h5.h5` is determined by an “oversampling” rate which can be set by the user in the script. This rate is called an “oversampling rate” because it represents how much more frequently than the Nyquist rate the data should be output, where the Nyquist rate is a sampling rate equal to twice the value of the highest frequency contained in the electromagnetic field solved for by the script. The OpenEMS script provides the reciprocal of the Nyquist rate in terms of a number of timesteps. For example, if the highest signal frequency is 120,000 Hz, then the Nyquist rate in Hertz is 240,000. The reciprocal of this is a time equal to approximately 4.167×10^{-6} s. If the OpenEMS time step is for example $\Delta t = 1.92583 \times 10^{-11}$ s, then the reciprocal of the Nyquist rate corresponds to 4.167×10^{-6} s / 1.92583×10^{-11} s = 216,356 such time steps. Hence, if the oversampling rate is set to 2, the script will output a dataset once for every $216356/2 = 108,178$ OpenEMS timesteps. That is, twice as frequently as the Nyquist rate in timesteps.

The actual choice of oversampling rate is nontrivial and must be made with the usefulness of the output in mind. Generally speaking, and particularly for early times in the simulation, the electric field specified at the mesh points of the volume by OpenEMS at any particular output time, cannot be depended upon to be quasi-static. But without a quasi-static electric field, it is not possible to compute an unambiguous value of voltage via line integration at any location within the volume, since to compute such an unambiguous voltage requires the computation to exhibit path-independence. We have addressed this issue by requiring that the output of the OpenEMS scripts must provide data at a sufficient rate to allow for time-averaging of the results over time intervals large enough to produce quasi-static time-averaged electric field components, yet small enough to provide information about the averaged field’s time-dependence over time intervals comparable to the timesteps required by the nerve and muscle modules’ integration schemes.

To make this clearer we provide an example. A typical timestep required by the nerve and muscle modules currently available as part of the HEMI Modeling and Simulation Software, is perhaps 2 microseconds whereas, a typical OpenEMS timestep might be as specified in our above example, namely, 1.92583×10^{-11} s. Suppose we wish to specify a time-averaged field once for (roughly) every 2 microseconds of time, such that the time-averaged field is the result of an average of D datasets of output by the OpenEMS script. To specify a time-averaged field at or near the time $t = 2.0 \times 10^{-6}$ s requires data sampled over the range $t \in (0.0 \text{ s}, 4.0 \times 10^{-6} \text{ s})$. Thus, we wish to set our oversampling rate, R_s , so that the OpenEMS script outputs D datasets for times covering this range. Let S be the number of OpenEMS timesteps needed to span the time interval over which we are averaging, (In the present case this time span is 4 microseconds), let R_s be the oversampling rate we seek, and let N_q be the Nyquist rate as expressed in OpenEMS timesteps (see the discussion above). Then one data set is output for every N_q/R_s OpenEMS timesteps. It follows that for S OpenEMS timesteps the number of datasets output will be given by Equation (15).

$$D = \frac{S}{N_q/R_s} \quad (15)$$

Recall, however, that we are specifying the value of D and seeking the required value of R_s ; thus, Equation (15) is rearranged into Equation (16).

$$R_s = \left(\frac{N_q}{S}\right) D \quad (16)$$

In our example, $S = 4.0 \times 10^{-6} \text{ s} / 1.92583 \times 10^{-11} \text{ s} \approx 207,704$ steps. So, if as above $N_q = 216,356$ steps, then for $D = 200$ datasets, we require the oversampling rate in Equation (17).

$$R_s = \left(\frac{216,356}{207,704}\right) \times 200 \approx 208 \quad (17)$$

Once this oversampling rate is determined, the total number of OpenEMS timesteps to be computed is determined in principle by how many time-averaged electric field data sets are required for the HEMI simulation. Note first however, that to get a time-averaged result for $t = 4.0 \times 10^{-6} \text{ s}$, requires only that we continue the simulation out to $t = 6.0 \times 10^{-6} \text{ s}$. Hence, we only require an additional 100 datasets to time average over the interval $t \in (2.0 \times 10^{-6} \text{ s}, 6.0 \times 10^{-6} \text{ s})$ since we have already generated the first 100 datasets required for the second time-averaged dataset in determining the first time-averaged dataset.

So, for example, if our HEMI simulation requires a duration of 1 millisecond with 2 microsecond timesteps, the total number of time-averaged datasets, excluding the $t = 0$ fields, is 500. Since unaveraged data will be required out to 1.002 milliseconds to obtain averaged data for $t = 1 \text{ ms}$, this implies a total number of required *unaveraged* datasets equal to $501 \times \frac{D}{2}$ where $D/2$ is the required number of *unaveraged* datasets per 2 microsecond-interval of time. In other words, in order to have, after averaging, time-averaged field data for all times in the range $t \in (0, 1.0 \times 10^{-3} \text{ s})$, at intervals of 2 microseconds, we require a number of OpenEMS timesteps sufficient to provide unaveraged datasets out to $1.002 \times 10^{-3} \text{ s}$. Now as determined above, a 2-microsecond time interval corresponds to $207,704/2 = 103,852$ OpenEMS timesteps. Hence for a HEMI simulation of duration 1 millisecond, we require a total of $103,852 \times 501 = 52,029,852$ OpenEMS timesteps.

We pause here to note how our example demonstrates two current limitations on the ability of the HEMI software to make use of the OpenEMS Electrical Dosimetry scripts. The number of OpenEMS timesteps required for generating sufficient information for a HEMI simulation is by itself prohibitive since OpenEMS simulations carried out over so many steps require very long run times. Additionally, the amount of data generated by such runs is invariably greater than can be stored on a desktop machine. Partly for these reasons, our current implementation is presented as a proof of concept rather than as a complete solution. In running tests of our FDTD voltage module, we have generated time-averaged electric field component datasets for a duration significantly smaller than the duration of the HEMI simulation itself, and for times greater than the largest time for which data is available the simulations simply cycle through the available

data. (This “cycling” is described in greater detail in section 9.1.1.1.4 below where we detail the actions of `get_ept`.)

5.2.3.2. FDTD Voltage

As stated previously, the FDTD voltage module—called “SEFDTDVoltage” in the HEMI source code, is designed to work with the output of the Electrical Dosimetry module discussed in the previous subsection. To run a HEMI simulation with the FDTD Voltage module, one first needs to generate an HDF5 formatted data file containing datasets for time-averaged electric field components at all mesh points within the volume of the test subject at regular time intervals small enough to ensure stability of the integrations carried out by the nerve and muscle modules that use the information provided by the voltage module. How such a data file is generated and formatted is the subject of the previous section.

The principal task of SEFDTDVoltage is to accept a set of coordinates provided to it by some other module (likely a nerve module) and to return the values of the external potential at those coordinates at a given time. The public interface of the SEFDTDVoltage module is as described above for the base class of voltage modules, where we recall that SEFDTDVoltage must implement an overriding version of the method called `EvaluateExternalPotentials`. The values of external potential SEFDTDVoltage provides are to be calculated on the basis of the time-averaged datasets of electric field components obtained from the Electrical Dosimetry module discussed previously. The principle internal data member of SEFDTDVoltage is a standard ordered map called `m_ept_ts`. The keys for this map are time values (represented as type `double`) and which correspond to times at which values of the external potentials are anticipated to be known and with values that are vectors of `doubles` which are meant to contain the values of the external potentials themselves at the time corresponding to the key and at the coordinates specified in the interface.

Details of the FDTD voltage model implementation are described in Appendix D.

5.3. Muscle Module

In this section we describe the structure of the muscle modules and the implementation of the only muscle model that is currently included in the HEMI M&S System (see Section 3.3).

5.3.1. Overview

Like the nerve modules discussed in section 5.1, muscle modules are presumed to implement mathematical models of muscle tissue which take the form of sets of coupled first order nonlinear ordinary differential equations. The public interface for muscle modules is therefore similar to the public interface for nerve modules. This interface is defined in the base class for muscle modules called `SEMuscle` whose definition is found in `SEMuscle.h`. In particular, we note the following elements of the definition:

- A `std::string` member `m_name` is used to distinguish among multiple instantiations of a muscle class. Associated with this member are the methods `SetName`, for setting the contents of `m_name`, `GetName` for querying the name of a particular muscle object, `HasName` for determining whether `m_name` is empty and `InvalidateName` for rendering `m_name` empty.

- A pure virtual function `ConfigureFromFile` is specified for muscle modules and must be implemented for modules deriving from `SEMuscle` according to the specific needs of the derived class. Serialization methods are provided for writing to and reading from files to facilitate input, output, and external storage and transmission of data members for muscles classes. As for the nerve base class, these methods—defined as pure virtual methods in `SEMuscle`—are called `SerializeToString`, `SerializeToFile`, `SerializeFromString` and `SerializeFromFile` and must be implemented by the developer of a muscle module deriving from `SEMuscle`.
- For time stepping, the pure virtual methods `SetCurrentTime`, `SetCurrentTimeIncrement`, and `TimeStep` are provided. All three of these methods are specified to take a single argument of type pointer to `SEScalarTime` in keeping with the requirement that public interfaces of modules specify all parameters required as arguments or return values in terms of objects belonging to the Common Data Model. Again, these methods must be overridden by implementations in modules deriving from the `SEMuscle` base class.

5.3.2. Wexler Muscle Module

Currently only one muscle module deriving from `SEMuscle` has been implemented. For mostly historical reasons⁵ this module is called `SEWexlerMuscle` (though its name may be changed to something more suitable in later releases of the HEMI software.) We leave presentation of the specifics of the mathematical models implemented in `SEWexlerMuscle` to section 3.3 and merely state here that numerical representations of differential equations for calcium dynamics, ATP production, and force generation are all implemented in `SEWexlerMuscle`. The implementation is analogous to that described for the `SESennNerve` nerve module. Like that module, the mathematical model implemented can be expressed in the form of Equation (18), where this time we have simply $M = 8$, with 6 of the equations describing calcium dynamics, one describing ATP production, and one describing force generation.

$$\frac{dy_i}{dt} = f(y_1 \cdots y_M; t) \quad (18)$$

The many parameters required by this model are all accounted for as protected data members of `SEWexlerMuscle` making use of the types available within the Common Data Model. Values for these parameters are set in the constructor for `SEWexlerMuscle`, but may also be modified and read in via a call to the module's `ConfigureFromFile` method which is implemented to read data in from a JSON formatted input file of the kind which is created upon calling `SEWexlerMuscle::SerializeToFile`.

⁵The first model implemented as a muscle module was the mathematical model of force described by (Wexler, Jun Ding, and Binder-Macleod 1997). The calcium model by (Otazu, Futami, and Hoshimiya 2001) was then added to the implementation to create the composite muscle model.

Integration time steps for this model are implemented in `SEWexlerMuscle` in a manner completely analogous to the manner in which they are implemented for the `SESennNerve` module. Like that module, The `TimeStep` method of `SEWexlerMuscle` is implemented to use the Boost `odeint` library's solver for ordinary differential equations called `runge_kutta_dopri5` which implements the Dormand-Prince Runge-Kutta method. As described above, this object has a method `do_step` which takes, as arguments, the vector of dependent variables y for the system of differential equations, a container for the current values of the right-hand side of the system of equations (represented as $f(y_1 \cdots y_M; t)$ in the above equation and called `dydt` in the code), the current time t , the current time increment dt , and a function object (called `wex` in `SEWexlerMuscle`) which provides the Dormand-Prince solver with a means to evaluate `dydt` at partial time steps during a full HEMI timestep dt . Using the information provided by these arguments, `do_step` carries out a single integration step and places the results in the vector y when `TimeStep` is called.

The function object `wex` is an instantiation of an internal class defined within the definition of `SEWexlerMuscle` and called `SEWexlerMuscle::Wexler` for which the parentheses operator is overloaded so that `wex` serves as a function object for evaluating `dydt` as described above. Indeed the implementations of the muscle model equations are all found within the implementation of `SEWexlerMuscle::Wexler::operator()`.

Most of the workings of `SEWexlerMuscle` are self-contained with the sole exception that this module does require an updateable value for the end-plate potential to be used in the model implemented in `SEWexlerMuscle::Wexler::operator()`. To accommodate this, a private data member `m_Vmt` is declared in the class declaration for `SEWexlerMuscle` and an accompanying public method `SetTransMembraneVoltage` is defined for assigning a value to this member. This member is updated via a call to `SetTransMembraneVoltage` made by the `Controller` object's `execute` method in the course of carrying out time stepping for a simulation. As currently implemented, the `Controller` object (see Section 5.4) obtains the value to assign to the `Wexler` muscle module's instance of `m_Vmt` from a call to the corresponding nerve module's `GetDistalNodeVoltage` method and provides this value as an argument to the muscle module's `SetTransMembraneVoltage` method. In this way, `m_Vmt` is updated for each time step prior to a call to `SEWexlerMuscle::TimeStep`. Note, that this implementation is actually incomplete since the transmembrane voltage of the distal nerve node and the end-plate potential are distinct quantities for which a module should be constructed providing the value of the latter given the value of the former. At present the two quantities are taken to be the same. In future release of the HEMI software this issue will be addressed.

5.4. Execution Control

As described above, modules for which time stepping must be implemented, such as the nerve and muscle modules, provision is made for time stepping within those modules only for one timestep per call to the modules' `TimeStep` methods. This is because the task of time stepping over a span of many timesteps is reserved for the `Controller` class. The philosophy adopted for the HEMI Modeling and Simulation software is that the individual modules for nerve, voltage, and muscle, should be as decoupled from one another as possible so that developers are

free to develop their own modules, and to mix and match them among other modules that have been developed for the software. Of particular significance, depending on the choice of mathematical model implemented by any of these modules, different modules may have different requirements for the size of time increments required for stable integration of their models. Some provision must be made to account for this in the design of the portion of the software that controls the execution of a simulation using these modules. For this and other reasons, the `Controller` class is defined to provide an object which serves to exert control over how the individual modules couple with one another during a simulation and over the details for how time stepping for a HEMI simulation is carried out.

The definition for the `Controller` class is found in `controller.h`. It consists of a relatively small number of protected data members, some associated public methods, a public `Initialize` method and a public `execute` method. We describe all these as follows:

- The `std::string` member called `m_working_directory` indicates the proper location for input and output files, and is initialized via an argument string to the constructor for a `Controller` object (See section 5.4).
- The three pointers of type `SEScalarTime` and named `m_global_time`, `m_global_time_increment`, and `m_global_simulation_duration` meant to hold values with meanings that are indicated by their variable names. They are “global” in the sense of defining those times associated with time stepping at the level of a HEMI global time step. They are so defined in anticipation of the possibility that a single global time step taken by the `Controller` may require multiple smaller time steps within one or more of the modules chosen. At present no such modules requiring this have been implemented, but this nomenclature has been adopted in anticipation of the need. These three data members are set using the public methods `SetGlobalTime`, `SetGlobalTimeIncrement` and `SetGlobalSimulationDuration` each of which takes a first argument of type `double` for the time’s value, and a second argument of type `TimeUnit` (defined within the Common Data Model) to specify what unit to associate with the value given for the time. (See section B.4 for a use example).
- A pointer to an object of type `SEVoltage` called `m_voltage`. As described in section 5.1, this pointer accesses the public interface for whatever voltage module has been chosen by the user to provide external potential values at locations needed by nerve modules. The voltage module is instantiated separately by the user, after which the user may assign `m_voltage` to point to the module instantiated via a call to the `Controller`’s public method `AddSEVoltage`.
- A `std::vector` called `m_nerves` whose elements are defined to be of type pointer to `SENerve`. This allows for the possibility of specifying multiple nerve objects within the simulation volume each of which can in principle be provided information about external potentials at the location of their specific nerve’s nodes from the voltage module pointed to by `m_voltage`. Nerve objects are instantiated separately by the user and pointers to them are added to `m_nerves` via a call to the `Controller`’s public method `AddSENerve`.

- A `std::vector` called `m_muscles` whose elements are defined to be of type pointer to `SEMuscle`. In this way, one can have one muscle object for each nerve object in a simulation. Muscle objects are instantiated separately by the user and pointers to them are added to `m_muscles` via call to the `Controller`'s public method `AddSEMuscle`.
- A public method, `Initialize()` which initializes time values local to the individual modules added to `m_nerves` and `m_muscles` and calls each of the `Initialize()` methods of the objects contained in `m_nerves` and `m_muscles`. It also checks that a valid voltage module has been instantiated and added to the `Controller` object, and calls its `Initialize()` method.
- A public method `execute()` which conducts time stepping for a simulation once all modules have been instantiated, configured, and initialized. `execute()` should be called by the user only after `Initialize()` has been called.

A private method `advanceTime()` which is called by `execute()` after each time step to update the current value of `m_global_time`.

6. Verification and Validation

The verification, validation, and accreditation (VV&A) for an integrated composite model is necessarily an ongoing process (Sokolowski and Banks 2010). The VV&A effort for the HEMI M&S system has been focused on specific aspects of the modeling framework under development and/or individual models. For example, V&V of the biomarker models has been conducted (Smith et al. 2019), and extensive verification of the SENN software conversion to C++ has been completed (Maxwell et al. 2020). An overview of the ongoing VV&A effort is provided in this section, and a more detailed V&V report is available in Appendix A.

6.1. Verification

Verification is defined as “[t]he process of determining that a model, simulation, or federation of models and simulations implementations and their associated data accurately represents the developer's conceptual description and specifications (DoD 2012).” Verification ensures that the implementation of models (both conceptual and logical) is as intended.

At this stage of the project, verification is possible. We are currently concerned with verifying the implementation of the constituent models as we integrate them into the composite model. Appendix A describes verification of the individual models that are included in the current version of the HEMI M&S system.

6.2. Validation

Validation is defined as “[t]he process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s) (DoD 2012).”

Validation is complicated, particularly considering that we are building a composite model. When models are integrated, the validity of the individual component models does not guarantee validity of the composite model. Adding to the complication is the that, according to the DoD definition, the perspective of the intended use (the reason the user is employing the model) must be known to perform validation. If the modeling objective is unknown, it is not possible to determine the metric with respect to which model fidelity is to be described, and therefore it is not possible to fully define the validity of the model.

Validity is related to credibility. The models that are being integrated in the HEMI M&S system have, for the most part, been peer reviewed and published, and thereby credibility is established. However, formal validation of the final integrated system must be performed to establish credibility of the composite model.

Comprehensive VV&A plans for the composite HEMI physiological modeling software are being developed along with the software system. In accordance with guidance provided by the sponsor (JIFCO 2020), we are, to the degree possible, following MIL-STD-3022 to develop these plans. We currently have planned sensitivity analysis and are considering other methods of uncertainty quantification. Additionally, we are investigating formal methods for quantify model alignment that may contribute to the validation (Young and Ezell 2019).

7. Outcomes Quantification

Recall the two questions that are driving the HEMI M&S system development.

- 1) What is the probability that a HEMI weapon exposure will result in significant injury?
- 2) To what degree in the HEMI weapon a direct causal factor if someone is injured during a HEMI weapon exposure?

Both are questions of causality.

At first glance, the first question may not seem like a causal question. However, while it is possible to quantify the risk of significant injury without consideration of causality, it is not possible to do so for novel weapons and exposure scenarios.

By contrast, the second question is clearly causal, and this question is arguably the most important in an operational context. Whether for acquisition, mission planning, or any other application, the operational context is certainly pertinent for anyone seeking to assess the risk of significant injury of HEMI weapons.

Simply put, warfighters need to know if a HEMI weapon will cause significant injury. Likewise, warfighters need to know if a weapon will cause incapacitation. Causality is key.

In this section, we review the risk quantification approach described in a previous report, and we argue that the previously described approach is necessary, but not sufficient, for quantifying outcomes from model and other data. We describe modifications to the previous strategy to create an integrated outcomes quantification strategy that involves computational modeling, empirical investigation, and structural causal modeling with application of modern causal inference methodologies. The integrated approach addresses the challenges of answering the causal questions.

7.1. Traditional Statistical Approach

Nonlethal weapons are required by the United States Department of Defense (DoD) to “minimize risk of fatalities, permanent injuries, or permanent damage to materiel; however, they shall not be required to have a zero probability of producing these effects in accordance with DoD Directive 3000.03E (Reference (d)) (Kendall 2018).” Although DoDI 3200.19 describes the general procedure for assessing risk, quantifying the RSI can be challenging. A methodology developed by the RSI Technical Working Group (Burgei et al. 2014) is in use by the DoD research group tasked with assessing RSI for new nonlethal weapon systems (Beier, Fleming, and Ashworth 2017), and we have adopted that methodology for this project.

The risk of significant injury (RSI) is a conditional probability based on the probability that an injury will occur, and the probability that an injury will be significant if it occurs (Burgei et al. 2014). The probability that an injury will occur may be computed using an injury risk curve that evaluates the risk of injury as a function of a specified metric that varies depending on the injury of interest. The probability that an injury is significant can be determined from available datasets as well as using the standard of care criteria (Comeaux and D’andrea 2012). Once the RSI is computed for each injury of concern, all of the RSIs may be aggregated to determine the overall risk of sustaining a significant injury from a HEMI device.

Quantification of the risk of significant injury due to HEMI weapon exposure is possible with data, and we described previously a mathematical approach for computing the risk of significant injury from the data (Maxwell et al. 2020).

There are several major challenges associated with using data (particularly observational data) to compute the risk of significant injury. One challenge is confounding. In our literature review we found that confounding was a significant issue in estimating the risk of any injury, and we presented rhabdomyolysis as an example (Pirone 2017). Figure 15, copied from the literature review, summarize some relevant causal factors of rhabdomyolysis. For the current effort, we are concerned only with the physiological causal pathways, but we note that some causal factors of rhabdomyolysis, e.g., drugs, could increase the likelihood of behaviors that influence other causal factors.

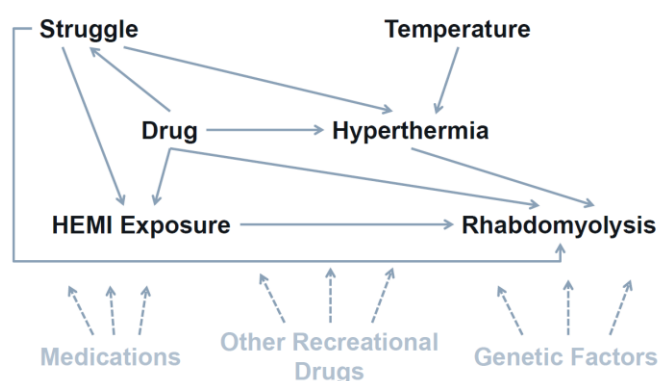


Figure 15. Causal complexity of rhabdomyolysis (Pirone 2017).

Others have also identified the challenge of using traditional methods for determining the health effects of HEMI weapons. As part of the Conducted Energy Weapons Strategic Initiative (CEWSI), the Canadian Academy of Health Sciences assembled a 14-member multidisciplinary panel of experts from Canada and abroad to answer three questions (Goudge et al. 2013).

- 1) What is the current state of scientific knowledge about the medical and physiological impacts of conducted energy weapons?
- 2) What gaps exist in the current knowledge about these impacts?
- 3) What research is required to close these gaps?

The panel had 5 key findings. First, each new CEW must be tested on its own to assess performance and the potential for adverse health effects. Second, a definitive assessment of risk is difficult due to a lack of high-quality data, but available studies suggest that significant injuries would be extremely rare. Third, isolating the causal factors that result in significant injury or death proximal to the application of a HEMI weapon is difficult because most exposure scenarios are complicated and involve multiple factors. Fourth, there are major gaps in knowledge, and the establishment of causal relationships is a priority. Fifth, addressing the knowledge gaps will require a series of integrated strategies, including novel approaches to computer modeling and animal studies.

Another challenge is that the occurrence of injury is drastically infrequent. In fact, previous work sponsored by JIFCO (formerly JNLWD) found very low risk of rhabdomyolysis (~1%), cardiac injury (~0.015%), respiratory dysfunction (0%), seizures (7%)⁶, and vertebral compression fractures (2.5%) (Comeaux and D’Andrea 2012). Similarly, for HEMI exposures lasting up to 30 seconds, a Human Effects Advisory Panel estimated the RSI for rhabdomyolysis to be zero, the RSI for cardiac events to be “near zero,” and the RSI for seizure to be zero (Kenny et al. 2012). A more recent review found that the human health risks of CEW exposure are low; however, it also found that the participants in the reviewed studies were not representative of populations that usually encounter CEWs making it difficult to draw conclusions regarding high-risk groups (Baliatsas et al. 2021).

These challenges arise because, if we rely on classical statistics, we are trying to answer causal questions from observational data. The only questions that we can answer with observational data are of the form, “If I see a HEMI weapon exposure, what is the probability that I will also see rhabdomyolysis?” The questions that we really want to answer are of the form, “If I use a HEMI weapon, what is the probability that rhabdomyolysis will occur?” They are also of the form, “If the HEMI weapon was not used, what is the probability that rhabdomyolysis have occurred anyway?”

It is possible to overcome these challenges using a randomized controlled trial (RCT). Judea Pearl, Turing Award-winning computer scientist who developed the modern causal inference methods described in the next section, calls the RCT “a man-made tool for uncovering the query $P(R|do(H))$, which is a property of nature (Pearl and Mackenzie 2018),” where $P(R|do(H))$ is the probability of rhabdomyolysis given a HEMI weapon exposure. With a RCT, the joint

⁶ When the probability of the probes striking the head is set to 7%.

distribution between R and H may be obtained by conditioning on the confounding variables (e.g., cocaine use).

However, to investigate HEMI mechanisms, a RCT would involve the random application of HEMI weapons to individuals and observing for rhabdomyolysis, a method that is costly and arguably unethical. RCTs using animal models have been conducted, and observed cases of rhabdomyolysis are exceedingly rare, but an exhaustive examination of the causal mechanisms of the naturally observed cases would require administration of cocaine and application of the weapon to unanesthetized animals (to elicit struggle). This approach would also be costly and unethical. What we need is a way to infer causality from observational data.

7.2. Modern Causal Inference

To describe how modern causal inference methods may be applied to quantify the probability of outcomes of HEMI weapon use, consider that Figure 15 may be reconstructed in the form of Figure 2, where drug use and other factors for which the cause is irrelevant to the model are lumped into the box labeled exogenous variables. Figure 16 shows the extension.

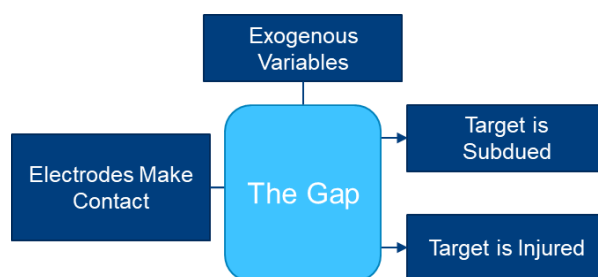


Figure 16. An extension of Figure 2 including exogenous variables.

Recall that we investigated the gap in Section 2. In doing so, we implicitly assumed the causal relationships between many key events that lead to both the intended and the unintended outcomes of interest. By explicitly defining the assumptions of causality, we can construct causal diagrams which are graphical representations of the structural causal models used in modern causal inference. For example, a notional causal diagram containing key physiological events of rhabdomyolysis is presented in Figure 17. Note that the diagram is constructed following the mechanisms described in Section 2.2.2, and it resembles Figure 10 which describes the “viscous cycle” of rhabdomyolysis.

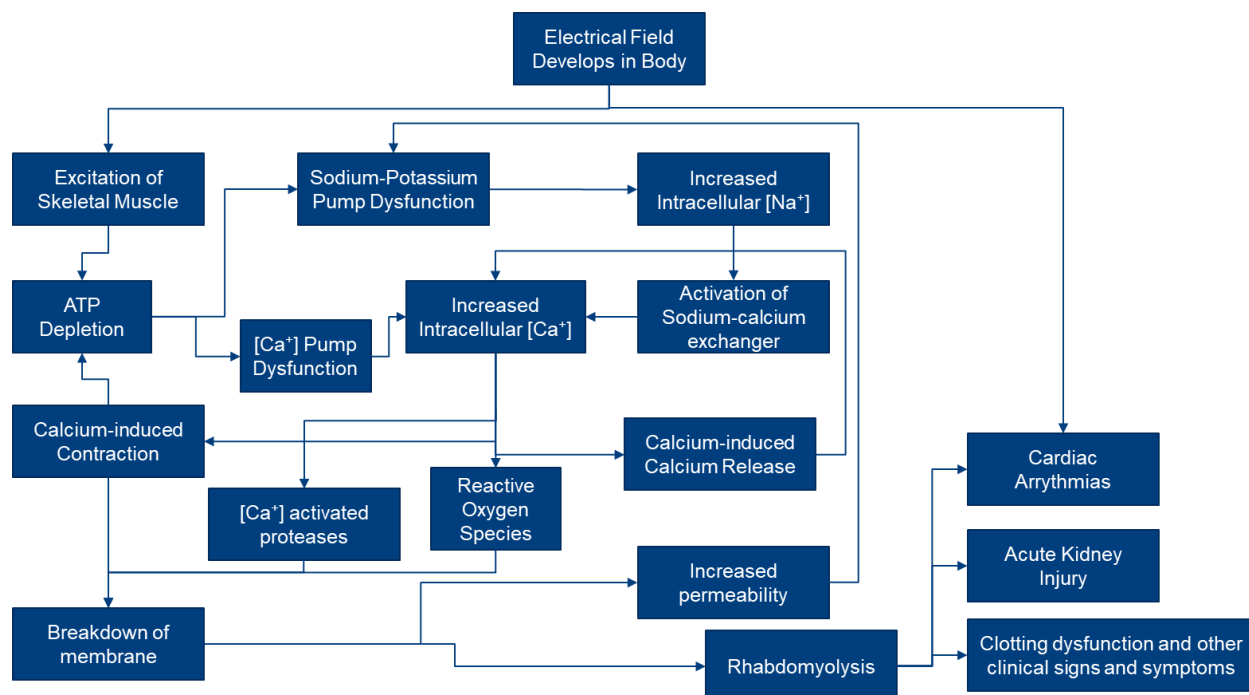


Figure 17. A notional causal diagram for rhabdomyolysis and acute renal injury.

The diagram is a graphical form of a structural causal model.

7.2.1. Structural Casual Modeling

A structural causal model is a tuple that contains endogenous variables (those for which causality is important in the model), exogenous variables (background variables, the cause of which are not important in the model), a set of functions describing the relationships between variables, and probability distribution over the exogenous variables.

Structural causal models allow investigation into causality beyond association and into intervention and counterfactuals. After building the causal diagrams, and thereby the structural causal models, we can use algorithmic methods to determine if the combination of data and assumptions are sufficient to answer a query (Pearl, Glymour, and Jewell 2019). In other words, we could use the methods of modern causal inference to identify variables for which data are needed given a specific query. We are currently solidifying the approach and expect to have a full description of the specific steps required early in the next phase.

8. Summary, Conclusions, and Possible Next Steps

In this report, we have summarized the work accomplished in the most recent phase of the HEMI physiology modeling project. We began with a brief introduction to the effort followed by an overview of the causal mechanisms of HEMI outcomes. Then, we described the mathematical models describing the HEMI physiological processes that were developed in this phase. Next, we provided an overview of our development approach and explained that the development methodology is driven by our desire to construct robust software infrastructure to facilitate interorganizational development of software tools that meet JIFCO and other military-specific

needs. We then detailed the software architecture and specific aspects of the implementation. We provided a brief V&V update in the main body of the report, and we provided additional detail in the V&V report that is included as Appendix A. Finally, an outcomes quantification approach was discussed.

To conclude the report, we discuss some of the future objectives for the program.

8.1. Near Future Objectives

There are three objectives that we intend to work towards in the next phase of the program. The first is to improve and continue to advance the computational modeling, including enhancement of the existing mathematical models, evaluation and potential integration of additional and/or alternative models, and the development and integration of the analysis and report modules into the HEMI M&S system. The second objective is to conduct formal structural modeling to build the necessary framework for identifying data requirements given a set of assumptions and a set of modeling objectives. The third objective is to begin to collaborate with an experimental partner.

8.1.1. Continue the Computational Modeling Effort

With the strong software infrastructure now in place, development time for the implementation and integration of mathematical models has been greatly reduced. In the next phase of the project, we will continue to improve the computational modeling. The top priority is to implement a model linking the intracellular calcium concentration to breakdown of the sarcolemma. The mechanism is well known. Normally, the intracellular calcium level is around 100 nM. When calcium increases 0.4-0.8 μM m-calpain becomes active, and at 3-50 μM u-calpain is activated. The increased intracellular calcium also activates phospholipases and triggers the production of reactive oxygen species. We are currently evaluating mathematical models of the processes for integration.

8.1.1.1. Future: Statistical Analysis Module

The SAM will be a class that defines the structure of models that provide statistical analysis, including traditional statistical inference and modern SCM-based causal inference.

8.1.1.2. Future: Metrics and Reports Module

The RM will be a class defining models that process results from any of the other modules into metrics that are relevant for the user and the use case. For example, an acquisition officer may need to know the risk of significant injury. A planning analyst may want to examine the time course of serum CK concentration to estimate medical requirements for an operation where intermediate force capabilities are being employed.

8.1.2. Formal Structural Modeling for Causal Inference

The notional causal diagram of rhabdomyolysis secondary to HEMI weapon exposure that is presented in Figure 17 was created using an informal process – that is, it was built by noting the implicit causal assumptions present in the system biology and mathematical models. We plan to use a more deliberate, formal conceptual model development approach using the Objects-Processes-Relations method (Turnitsa 2012). Objects are persistent components of a model that maintain identity over time. They are artifacts of the system and hold quantitative and/or

qualitative value(s) describing their individual states. Ontologically, objects are things that exist. By contrast, Processes are, ontologically, things that occur. Processes are the dynamic behavior elements of models that mark a change in the state of one or more Objects, and they represent the causality that must be captured in models for an agent to make credible, meaningful plans. Relations are the linkages between two other conceptual elements of the model.

After the formal conceptual model is complete, we will build structural causal models and the associated directed acyclic graphs. Then, we will be ready to apply modern causal inference methods to identify data requirements for specific modeling objectives.

8.1.3. Collaborate with Empiricists

We will conduct the formal causal modeling and analysis described above at the beginning of the next phase of development. We are hoping to execute the process in collaboration with an empirical partner. That way, if we identify necessary data given a set of assumptions that cannot be obtained experimentally, we can adjust the assumptions accordingly.

9. References

- Ajeet, Ajeet. 2008. “Using the ADI-FDTD Method to Compute Currents Induced in the Human Body by HEMI Devices at Low Frequencies.” Raleigh: North Carolina State University.
- Bagur, Rafaela, and György Hajnóczky. 2017. “Intracellular Ca²⁺ Sensing: Its Role in Calcium Homeostasis and Signaling.” *Molecular Cell* 66 (6): 780–88. <https://doi.org/10.1016/j.molcel.2017.05.028>.
- Baliatsas, Christos, Jenny Gerbecks, Michel L. A. Dückers, and C. Joris Yzermans. 2021. “Human Health Risks of Conducted Electrical Weapon Exposure: A Systematic Review.” *JAMA Network Open* 4 (2): e2037209. <https://doi.org/10.1001/jamanetworkopen.2020.37209>.
- Beier, Eric Benjamin, Jimmy Fleming, and Alan Ashworth. 2017. “Assessing Risk of Significant Injury from Multiple Stimuli Engagements.” AFRL-RH-FS-PC-2017-0001.
- Belcastro, A. N. 1993. “Skeletal Muscle Calcium-Activated Neutral Protease (Calpain) with Exercise.” *Journal of Applied Physiology* 74 (3): 1381–86. <https://doi.org/10.1152/jappl.1993.74.3.1381>.
- Belcastro, Angelo N., Leann D. Shewchuk, and Daniel A. Raj. 1998. “Exercise-Induced Muscle Injury: A Calpain Hypothesis.” *Molecular and Cellular Biochemistry* 179 (1–2): 135–45. <https://doi.org/10.1023/A:1006816123601>.
- Berchtold, Martin W., Heinrich Brinkmeier, and Markus Müntener. 2000. “Calcium Ion in Skeletal Muscle: Its Crucial Role for Muscle Function, Plasticity, and Disease.” *Physiological Reviews* 80 (3): 1215–65. <https://doi.org/10.1152/physrev.2000.80.3.1215>.
- Berger, David. 2020. “Intermediate Force Capabilities: Bridging the Gap Between Presence and Lethality (Executive Agent’s Planning Guidance 2020).” U.S. DoD.

- Bono, Bernard de, Pierre Grenon, and Stephen John Sammut. 2012. "ApiNATOMY: A Novel Toolkit for Visualizing Multiscale Anatomy Schematics with Phenotype-Related Information." *Human Mutation* 33 (5): 837–48. <https://doi.org/10.1002/humu.22065>.
- Boron, Walter F., ed. 2012. *Medical Physiology: A Cellular and Molecular Approach*. Updated 2. ed. Philadelphia, Pa: Saunders, Elsevier.
- Bosch, Xavier, Esteban Poch, and Josep M Grau. 2009. "Rhabdomyolysis and Acute Kidney Injury." *New England Journal of Medicine* 361 (1): 62–72.
- Bruce, Jason IE. 2010. "Plasma Membrane Calcium Pump Regulation by Metabolic Stress." *World Journal of Biological Chemistry* 1 (7): 221. <https://doi.org/10.4331/wjbc.v1.i7.221>.
- Bruce, Jason I.E. 2018. "Metabolic Regulation of the PMCA: Role in Cell Death and Survival." *Cell Calcium* 69 (January): 28–36. <https://doi.org/10.1016/j.ceca.2017.06.001>.
- Brumback, Roger A., Daniel L. Feeback, and Richard W. Leech. 1995. "Rhabdomyolysis Following Electrical Injury." In *Seminars in Neurology*, 15:329–34. \copyright 1995 by Thieme Medical Publishers, Inc. <https://www.thieme-connect.com/products/ejournals/pdf/10.1055/s-2008-1041040.pdf>.
- Comeaux, James A., and John D'Andrea. 2012. "Risk and Effects Assessment of Electro-Muscular Incapacitation (EMI) Technologies." Assessment Report NAMRU-SA TR-2012-01. Naval Medical Research Unit-San Antonio.
- Cooks, Tammi, T. Keefe Davis, June Hu, Rachel Metheny, Michael Schwartz, and Roy Gerona. 2016a. "'Smoking' Guns: Answers." *Pediatric Nephrology* 31 (1): 63–66. <https://doi.org/10.1007/s00467-014-2983-4>.
- . 2016b. "'Smoking' Guns: Questions." *Pediatric Nephrology* 31 (1): 61–62. <https://doi.org/10.1007/s00467-014-2978-1>.
- Dawes, Donald M., Jeffrey D. Ho, Robert F. Reardon, Seth R. Strote, Rebecca S. Nelson, Erik J. Lundin, Benjamin S. Orozco, Sebastian N. Kunz, and James R. Miner. 2011. "The Respiratory, Metabolic, and Neuroendocrine Effects of a New Generation Electronic Control Device." *Forensic Science International* 207 (1–3): 55–60. <https://doi.org/10.1016/j.forsciint.2010.08.028>.
- Despa, Florin, Suki Basati, Zhen-Du Zhang, John D'Andrea, J Patrick Reilly, Elena N. Bodnar, and Raphael C. Lee. 2009. "Electromuscular Incapacitation Results from Stimulation of Spinal Reflexes." *Bioelectromagnetics* 30 (5): 411–21. <https://doi.org/10.1002/bem.20489>.
- DoD. 2012. "DoD Standard Practice: Documentation of Verification, Validation, and Accreditation (VV&&A) for Models and Simulations (MIL-STD-3022)." <https://vva.msco.mil/>.
- Fredsted, Anne, Hanne Gissel, Klavs Madsen, and Torben Clausen. 2007. "Causes of Excitation-Induced Muscle Cell Damage in Isometric Contractions: Mechanical Stress or Calcium Overload?" *American Journal of Physiology - Regulatory Integrative and Comparative Physiology* 292 (6): 2249–58. <https://doi.org/10.1152/ajpregu.00415.2006>.

- Gissel, H. 2005. "The Role of Ca²⁺ in Muscle Cell Damage." *Annals of the New York Academy of Sciences* 1066 (1): 166–80. <https://doi.org/10.1196/annals.1363.013>.
- Gissel, Hanne. 2005. "The Role of Ca²⁺ in Muscle Cell Damage." *Annals of the New York Academy of Sciences* 1066: 166–80. <https://doi.org/10.1196/annals.1363.013>.
- Gleason, Dr. James Benjamin. 2015. "TASER ® Electronic Control Device- Induced Rhabdomyolysis and Renal Failure: A Case Report." *JOURNAL OF CLINICAL AND DIAGNOSTIC RESEARCH*. <https://doi.org/10.7860/JCDR/2015/15465.6608>.
- Goudge, Mark, James Brophy, George Carruthers, Igor Efimov, Derek Exner, Robert Gordon, Christine Hall, et al. 2013. "The Health Effects of Conducted Energy Weapons."
- Gross, Erica R., Joseph Portericko, and M. D. D'Andrea Joseph. 2013. "Rhabdomyolysis and Oliguric Renal Failure after Use of TASER®: Is It Really Safe?" *The American Surgeon* 79 (12): E337.
- Hall, John E., and Arthur C. Guyton. 2011. *Textbook of Medical Physiology*. 12th ed. Philadelphia: Saunders.
- Haskins, Paul A. n.d. "Conducted Energy Devices: Policies on Use Evolve to Reflect Research and Field Deployment Experience," 9.
- Ho, Jeffrey D., Donald M. Dawes, Robert F. Reardon, Seth R. Strote, Sebastian N. Kunz, Rebecca S. Nelson, Erik J. Lundin, Benjamin S. Orozco, and James R. Miner. 2011. "Human Cardiovascular Effects of a New Generation Conducted Electrical Weapon." *Forensic Science International* 204 (1–3): 50–57. <https://doi.org/10.1016/j.forsciint.2010.05.003>.
- Huang, Jiaru, and Xiaoping Zhu. 2016. "The Molecular Mechanisms of Calpains Action on Skeletal Muscle Atrophy." *Physiological Research* 65 (4): 547–60. <https://doi.org/10.33549/physiolres.933087>.
- JIFCO. 2020. "JIFCO Guidance: Development and Documentation for Modeling & Simulation."
- Kendall, Frank. 2018. "DOD Instruction Number 3200.19: Non-Lethal Weapons (NLW) Human Effects Characterization."
- Kenny, John, Cameron Bass, William Bozeman, Viktor Bovbjerg, and Donald Farrer. 2012. "The Human Effects Advisory Panel's Assessment of The HEMI RSI and Efficacy Study." JNLW12-075. Penn State University Applied Research Laboratory.
- Kim, Jooyoung, Joohyung Lee, Sojung Kim, Ho Young Ryu, Kwang Suk Cha, and Dong Jun Sung. 2016. "Exercise-Induced Rhabdomyolysis Mechanisms and Prevention: A Literature Review." *Journal of Sport and Health Science* 5 (3): 324–33. <https://doi.org/10.1016/j.jshs.2015.01.012>.
- Kohl, P, E J Crampin, T A Quinn, and D Noble. 2010. "Systems Biology: An Approach." *Clinical Pharmacology & Therapeutics* 88 (1): 25–33. <https://doi.org/10.1038/clpt.2010.92>.

- Kroll, M. W., M. A. Brave, H. M. O. Pratt, K. K. Witte, S. N. Kunz, and R. M. Luceri. 2019. "Benefits, Risks, and Myths of TASER® Handheld Electrical Weapons." *Human Factors and Mechanical Engineering for Defense and Safety* 3 (1): 7. <https://doi.org/10.1007/s41314-019-0021-9>.
- MacLennan, David H., William J. Rice, and N. Michael Green. 1997. "The Mechanism of Ca²⁺ Transport by Sarco(Endo)Plasmic Reticulum Ca²⁺-ATPases." *Journal of Biological Chemistry* 272 (46): 28815–18. <https://doi.org/10.1074/jbc.272.46.28815>.
- Maier, Andrew, Patricia Nance, Paul Price, Clifford J. Sherry, J. P. Reilly, B. J. Klauenberg, and Jonathan T. Drummond. 2005. "Human Effectiveness and Risk Characterization of the Electromuscular Incapacitation Device - A Limited Analysis of the TASER. Part 1. Technical Report." Fort Belvoir, VA: Defense Technical Information Center. <https://doi.org/10.21236/ADA434044>.
- Maier, Andrew, Patricia Nance, Paul Price, Clifford J. Sherry, and J. Patrick Reilly. 2005. "Human Effectiveness and Risk Characterization of the Electromuscular Incapacitation Device-A Limited Analysis of the TASER. Part 2. Appendices." DTIC Document. <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA434298>.
- Maxwell, Kidist, Tim Dennis, Camille Romanowski, Rodney Metoyer, and Stephen Beaulieu. 2020. "Summary Report for: Human Electro-Muscular Incapacitation (HEMI) Weapons: Physiological Modeling."
- Metoyer, Rodney. 2017. "HEMI Project Phase 1 Recap, Progress Report, and Discussion." Quantico, VA, June 2.
- Momeni, Hamid Reza. 2011. "Role of Calpain in Apoptosis." *Cell Journal* 13 (2): 65–72.
- Osanlouy, Mahyar, Anita Bandrowski, Bernard de Bono, David Brooks, Antonino M. Cassarà, Richard Christie, Nazanin Ebrahimi, et al. 2021. "The SPARC DRC: Building a Resource for the Autonomic Nervous System Community." *Frontiers in Physiology* 12 (June): 27. <https://doi.org/10.3389/fphys.2021.693735>.
- Otazu, Gonzalo H., Ryoko Futami, and Nozomu Hoshimiya. 2001. "A Muscle Activation Model of Variable Stimulation Frequency Response and Stimulation History, Based on Positive Feedback in Calcium Dynamics." *Biological Cybernetics* 84 (3): 193–206. <https://doi.org/10.1007/s004220000207>.
- Panescu, Dorin, Mark W. Kroll, Igor R. Efimov, and James D. Sweeney. 2006. "Finite Element Modeling of Electric Field Effects of TASER Devices on Nerve and Muscle." In *Engineering in Medicine and Biology Society, 2006. EMBS'06. 28th Annual International Conference of the IEEE*, 1277–79. IEEE. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4461992.
- Park, Eun-Jung, Sang-Cheon Choi, Jung-Hwan Ahn, and Young-Gi Min. 2013. "Repetitive TASER X26 Discharge Resulted in Adverse Physiologic Events with a Dose-Response Relationship Related to the Duration of Discharge in Anesthetized Swine Model." *Journal of Forensic Sciences* 58 (1): 179–83. <https://doi.org/10.1111/j.1556-4029.2012.02287.x>.

- Pasquier, Mathieu, Pierre-Nicolas Carron, Laurent Vallotton, and Bertrand Yersin. 2011. "Electronic Control Device Exposure: A Review of Morbidity and Mortality." *Annals of Emergency Medicine* 58 (2): 178–88. <https://doi.org/10.1016/j.annemergmed.2011.01.023>.
- Pearl, Judea, Madelyn Glymour, and Nicholas P. Jewell. 2019. *Causal Inference in Statistics: A Primer*. Reprint with revisions. Chichester: Wiley.
- Pearl, Judea, and Dana Mackenzie. 2018. *The Book of Why: The New Science of Cause and Effect*. Basic Books.
- Pirone, Jason. 2017. "ARA HEMI Program Post Literature Review Briefing." Quantico, VA, February 10.
- Powers, Scott K., Li Li Ji, Andreas N. Kavazis, and Malcolm J. Jackson. 2011. "Reactive Oxygen Species: Impact on Skeletal Muscle." *Comprehensive Physiology* 1 (2): 941–69. <https://doi.org/10.1002/cphy.c100054>.
- Price, Owen, Bahman Asgharian, and Jacob Bellman. 2018. "Human Electro-Muscular Incapacitation (HEMI): Physiological Modeling Weapons." ARA/HEMI-FR-18-007-A. Joint Non-Lethal Weapons Directorate: Applied Research Associates, Inc.
- Reilly, J Patrick. 1988. "Electrical Models for Neural Excitation Studies." *Johns Hopkins APL Technical Digest* 9 (1): 44–59.
- Reilly, J. Patrick. 1998. *Applied Bioelectricity: From Electrical Stimulation to Electropathology*. New York: Springer.
- Reilly, J Patrick. 2016. "Survey of Numerical Electrostimulation Models." *Physics in Medicine and Biology* 61 (12): 4346–63. <https://doi.org/10.1088/0031-9155/61/12/4346>.
- Reilly, J Patrick, and Alan M Diamant. 2011. *Electrostimulation: Theory, Applications, and Computational Model*. Artech House.
- Rockenfeller, Robert, Michael Günther, Norman Stutzig, Daniel F. B. Haeufle, Tobias Siebert, Syn Schmitt, Kay Leichsenring, Markus Böhl, and Thomas Götz. 2020. "Exhaustion of Skeletal Muscle Fibers Within Seconds: Incorporating Phosphate Kinetics Into a Hill-Type Model." *Frontiers in Physiology* 11 (May): 306. <https://doi.org/10.3389/fphys.2020.00306>.
- Sanford, Janyce M., Gregory J. Jacobs, Edward J. Roe, and Thomas E. Terndrup. 2011. "Two Patients Subdued with a TASER® Device: Cases and Review of Complications." *The Journal of Emergency Medicine* 40 (1): 28–32. <https://doi.org/10.1016/j.jemermed.2007.10.059>.
- Shorten, Paul R., Paul O'Callaghan, John B. Davidson, and Tanya K. Soboleva. 2007. "A Mathematical Model of Fatigue in Skeletal Muscle Force Contraction." *Journal of Muscle Research and Cell Motility* 28 (6): 293–313. <https://doi.org/10.1007/s10974-007-9125-6>.

- Smith, Krista, Owen Price, Bahman Asgharian, David Crary, Aaron Parks, Nicole Flaig, Stephen Beaulieu, and Stephen Beaulieu. 2019. "Human Electro-Muscular Incapacitation (HEMI): Physiological Modeling Weapons." ARA/HEMI-FR-19-007-A. Joint Non-Lethal Weapons Directorate: Applied Research Associates, Inc.
- Sokolowski, John A., and Catherine M. Banks, eds. 2010. *Modeling and Simulation Fundamentals: Theoretical Underpinnings and Practical Domains*. Hoboken, N.J: John Wiley.
- Turnitsa, Charles Daniel. 2012. "Exploring the Components of Dynamics Modeling Techniques." PhD, Old Dominion University.
- Tyagi, Aaron, Alex Gill, and Brent Felton. 2017. "Thoracic Compression Fracture as a Result of Taser® Discharge." *Clinical Practice and Cases in Emergency Medicine* 1 (4): 319–22. <https://doi.org/10.5811/cpcem.2017.7.33508>.
- Vanderklisch, Peter W., and Ben A. Bahr. 2000. "The Pathogenic Activation of Calpain: A Marker and Mediator of Cellular Toxicity and Disease States." *International Journal of Experimental Pathology* 81 (5): 323–39. <https://doi.org/10.1111/j.1365-2613.2000.00169.x>.
- Wexler, A.S., Jun Ding, and S.A. Binder-Macleod. 1997. "A Mathematical Model That Predicts Skeletal Muscle Force." *IEEE Transactions on Biomedical Engineering* 44 (5): 337–48. <https://doi.org/10.1109/10.568909>.
- Young, John B., and Barry Ezell. 2019. "Validation Risk in Hierarchical Models." In *2019 Winter Simulation Conference (WSC)*, 3086–97. National Harbor, MD, USA: IEEE. <https://doi.org/10.1109/WSC40007.2019.9004868>.
- Zhou, Jingsong, Kamal Dhakal, and Jianxun Yi. 2016. "Mitochondrial Ca²⁺ Uptake in Skeletal Muscle Health and Disease." *Sci China Life Sci.* 59 (8): 770–76. <https://doi.org/10.1007/s11427-016-5089-3>.
- Zipes, Douglas P. 2014. "TASER Electronic Control Devices Can Cause Cardiac Arrest in Humans." *Circulation* 129 (1): 101–11.

Appendix A. V&V Report

Verification is defined as “[t]he process of determining that a model, simulation, or federation of models and simulations implementations and their associated data accurately represents the developer’s conceptual description and specifications (DoD 2012).” Verification ensures that the implementation of models (both conceptual and logical) is as intended.

Validation is defined as “[t]he process of determining the degree to which a model, simulation, or federation of models and simulations, and their associated data are accurate representations of the real world from the perspective of the intended use(s) (DoD 2012).”

According to this definition, the modeling objective (the reason the user is employing the model) must be known in order to perform validation. If the modeling objective is unknown, it is not possible to determine the metric with respect to which model fidelity is to be described.

At this stage of the project, verification is possible. Validation is complicated, particularly considering that we are building a composite model. When models are integrated, the validity of the individual component models does not guarantee validity of the composite model.

We are currently concerned with verifying the implementation of the constituent models as we integrate them into the composite model. We are also concerned with conveying the credibility of the component models, but validation of the composite model will not be possible until all models are integrated, and the modeling objectives are established. This report describes verification of the individual models that are included in the current version of the HEMI M&S system.

A.1. SENN Nerve Model Verification

Verification of the ObjSENN implementation of the SENN nerve model was described in a previous report (Maxwell et al. 2020). An overview of the verification results is provided here for convenience.

We performed 14 simulations using the ObjSENN implementation and compared the results to results of simulations conducted using the SENN software that is available from the publisher of (Reilly and Diamant 2011). We previously compared results from the binary distribution to the compiled source code and found compiler-dependent differences in the results, thus we use the binary distribution results as the basis for comparison.

All 14 simulations performed for verification are described in (Reilly and Diamant 2011), and one is described in considerable detail in Chapter 14. The input parameters for this case are listed in Table 1.

Table 1. Test Case 1 input parameters.

SINETEST	CM=2.0,	XA=100.0,	DELAY=0.0,	TAUS=0.0,	ITHR=0,
&FIBER	GM=30.365,	YA=100.0,	FREQ=5.0,	VREF=0.0,	VTH=80.0,
NNODES=51,	RHOI=110.0,	WIREL=0.85,	PHASE=0.0,	NP=1,	NTHNODE=3,
NLIN1=1,	RHOE=300.0,	IWAVE=2,	FREQ2=0.0,	FS=0,	DELT= 0.0002,
NLIN2=51,	&END	UIO=16.0,	PHASE2=0.0,	S=1,	DELT2M=4.0,
NODE1=25,	&STIMULUS	XPD=2.0,	AMP2=0.0,	NTRP=1,	FINAL=4.0,
DIAM=0.0020,	XC=0.0,	UIO2=0.0,	NSINES=27,	&END	IPRNT=0,
GAP=0.00025,	YC=1.0,	XPD2=0.0,	DCOFF=0.0,	&CONTROL	&END

In this case, a simple sinusoidal stimulus is applied and the SENN software is executed with the “threshold seeking” mode inactive (where the stimulus magnitude is adjusted to find a threshold for stimulation of the nerve), thus, the simulation time is allowed to continue even after an action potential is observed. By contrast, most of the other tests have threshold seeking mode active, and therefore the simulation time stops when an action potential is observed, and the simulation is reset for the next iteration in the threshold seeking routine.

The results of the first case are shown in the following subsection, and the overall results of the nerve verification testing are described in the subsequent subsection.

A.1.1. Results of Nerve Verification Case 1

The maximum percent difference between all values in the data.out files produced by the Fortran version and the C++ version are summarized in Table 2.

Table 2. Test Case 1 maximum percent difference between Fortran and C++ implementations.

Value	Maximum Percent Difference
Time	0.000
Transmembrane Potential	0.003
SUMK	0.219
DERY	0.276
TIM	0.191

Figure 18 shows the transmembrane voltage at the nodes for which data are printed in data.out (nodes 25-34) for the C++ version (left) and the Fortran version (right).

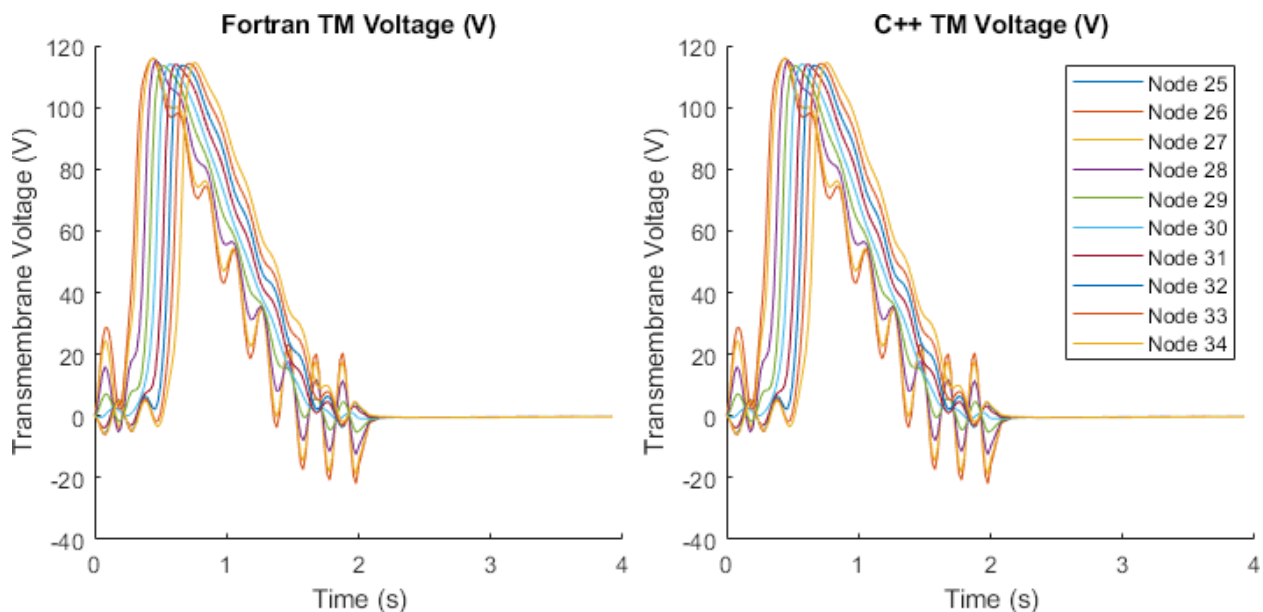


Figure 18. Transmembrane potential of select nodes for Test Case 1.

Figure 19 shows the transmembrane voltage of the print node as computed by the Fortran implementation and the C++ implementation of SENN for Test Case 1, demonstrating that the two implementations are in agreement with respect to the voltage of that one node.

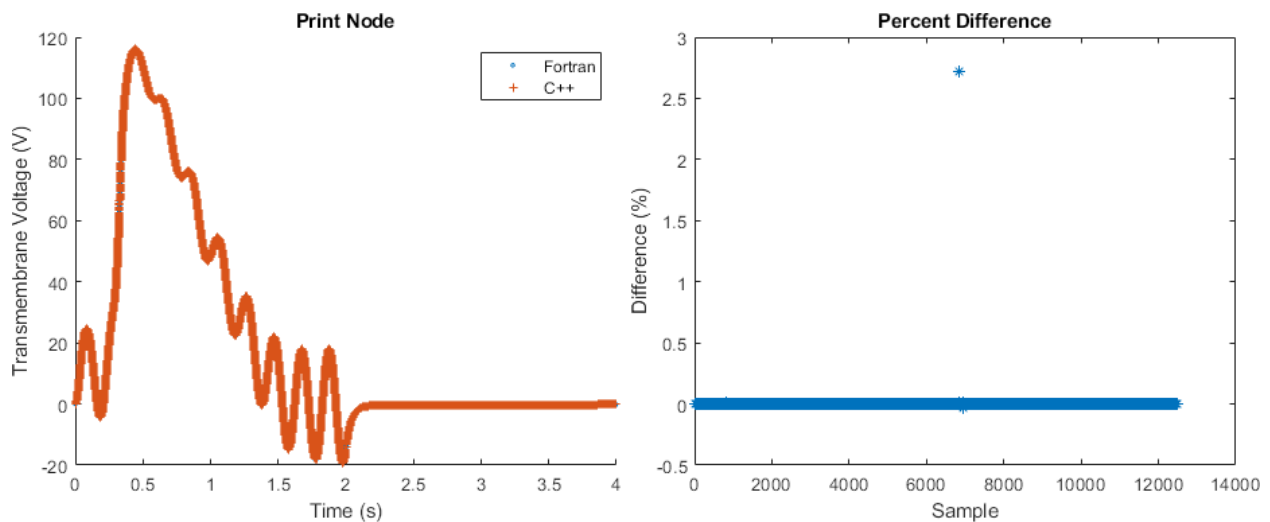


Figure 19. Transmembrane voltage of the print node from the C++ and Fortran implementations for Test Case 1 (left) and the percent difference between them (right).

Figure 20 shows the transmembrane voltage of the highest-voltage node as computed by the Fortran implementation and the C++ implementation of SENN for Test Case 1. The figure demonstrates that both versions predict an excitation at the same time.

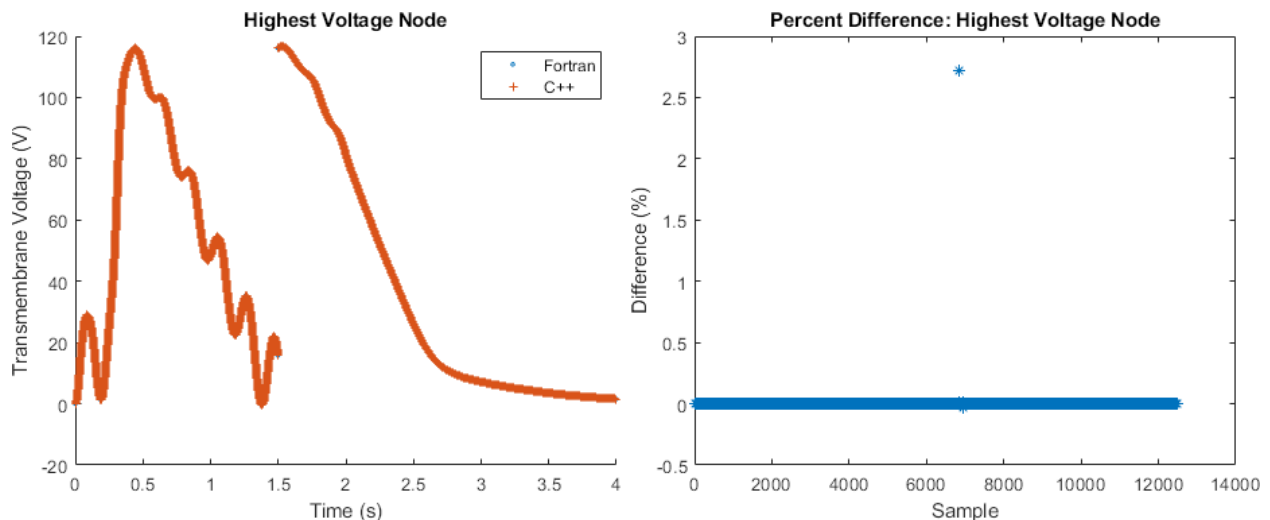


Figure 20. Transmembrane voltage of the highest-voltage node from the C++ and Fortran implementations for Test Case 1 (left) and the percent difference between them (right).

A.1.2. Nerve Verification Results Conclusions

Results of the other 13 verification tests are described in detail in (Maxwell et al. 2020) and summarized in Table 3. Note that the verification was performed using all data available from the original SENN binary executable, which includes the data.out file, the fort.17 file, and the fort.30 file. Information about what data are represented in these files is available in chapter 14 of (Reilly and Diamant 2011).

The percent difference is computed using the following formula.

$$\left| \frac{ObjSenn\ Value - SENN\ Value}{SENN\ Value} \right|$$

Note that the formula results in deceptively large values when the original SENN value of a data point is very small. For the transmembrane potential only, we eliminate any percent difference that is more than ten standard deviations from the mean of the data set. We have not yet performed this outlier rejection for all outputs, thus some “debugging outputs,” particularly the SUMK output, show large percent differences. However, we note that the transmembrane potential is the output of interest for the composite model. Furthermore, no output from the data.out file, other than the SUMK output, showed a percent difference greater than 0.25 percent.

Table 3. Nerve model verification summary data.

Test Number	Maximum % Difference (Transmembrane Potential)	Maximum % Difference (SUMK)
1	0.025	0.276
2	0.175	728.0
3	0.190	980.6
4	0.085	872.4
5	0.119	417.2
6	1.739	639.6
7	0.133	231.5
8	1.1361	86.23
9	0.207	393.4
10	0.164	1.163
11	0.252	0.837
12	3.258	3.258
13	1.162	64.49
14	15.38	418.6

A.2. Muscle Model Verification

At this stage of the HEMI M&S system development, muscle model verification involved comparing the results of the implemented Otazu model to the results published in (Otazu, Futami, and Hoshimiya 2001). We conducted three verification tests. The first verification test compares the HEMI M&S muscle model results to those presented in Figure 3A of (Otazu, Futami, and Hoshimiya 2001). The second test compares to Figure 3B, and the third test compares to Figure 6A of the same publication.

A.2.1. Muscle Verification Test 1

The first verification test is the simulation of potentiation performed by Otazu et al., the results of which are presented in Figure 3A of (Otazu, Futami, and Hoshimiya 2001) which is reproduced here as Figure for convenience. A train of half-sine shaped pulses at a frequency of 9 Hz is applied for two seconds. After the 2-second stimulation, the stimulus is removed for two seconds of simulation time, then a 50 Hz stimulation is applied for two more seconds. Following the second stimulation, the equilibrium concentration of calcium is elevated, a phenomenon called potentiation that occurs when the concentration of inactivation sites bound to calcium exceeds the bifurcation value (see Section 3.3). A subsequent stimulation at 9 Hz applied 4 seconds after the 50 Hz stimulation produces a greater mean calcium concentration and thus a greater fraction of activated troponin (and thereby more force).

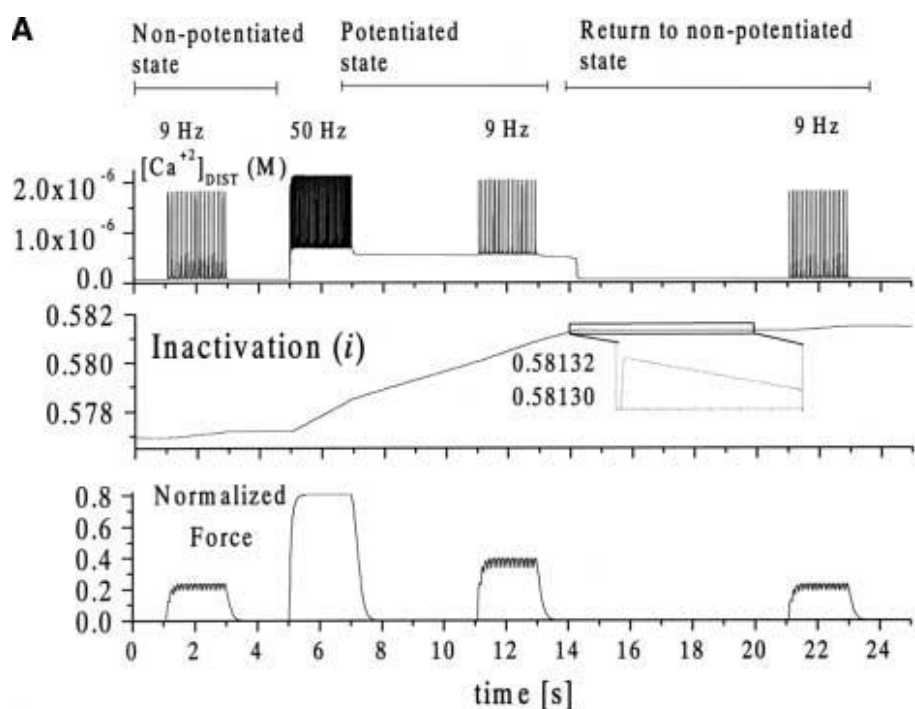


Figure 21. Figure 3A from (Otazu, Futami, and Hoshimiya 2001).

The results from the simulation using the HEMI M&S muscle model for distal calcium concentration are shown below in Figure .

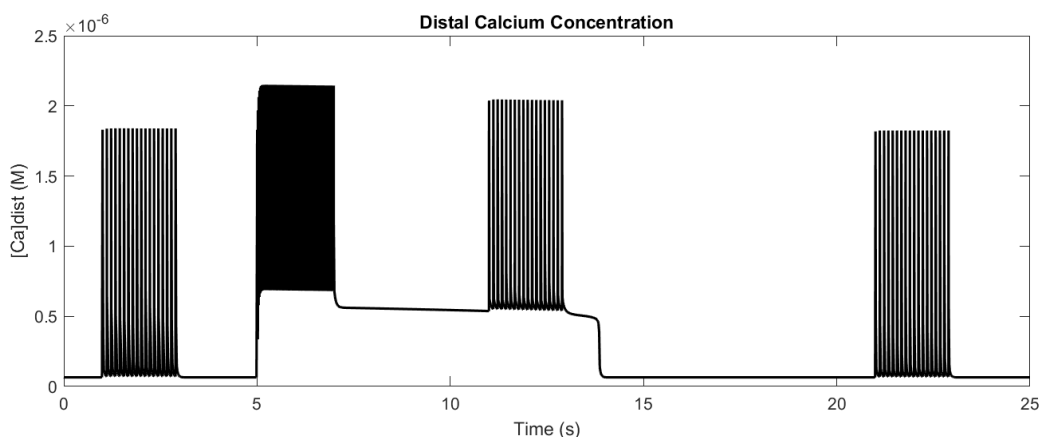


Figure 22. Distal calcium concentration results from the first verification test.

Figure was digitized for a quantitative comparison of the distal calcium, as shown in Table 4.

Table 4. Distal calcium concentrations at select times during simulation.

Event	Otazu Value	HEMI M&S Value
Before initial stimulus	6.12e-8	6.52e-8
Minimum during 1 st stimulus	7.55e-8	7.19e-8
Maximum during 1 st stimulus	1.83e-6	1.84e-6
After 1 st stimulus and before 2 nd	5.73e-8	6.52e-8
Minimum during 2 nd stimulus	6.79e-7	6.84e-7
Maximum during 2 nd stimulus	2.13e-6	2.12e-6
After 2 nd stimulus and before 3 rd	5.65e-7	5.49e-7

Minimum during 3 rd stimulus	5.63e-7	5.45e-7
Maximum during 3 rd stimulus	2.05e-6	2.04e-6
End resting	7.81e-8	6.45e-8

Taken together, Figure and Figure qualitatively verify the implementation of the Otazu model in the HEMI M&S system. The comparison of the digitized Otazu data to the HEMI M&S implementation in Table 4 provides quantitative support.

A.2.2. Muscle Verification Test 2

The second test was a comparison of the results of a simulation of potentiation, like the first verification test, but with an overstimulation state induced by a long 50 Hz stimulation that forces the muscle past the potentiated state. The results of this simulation are shown in Figure 3B in (Otazu, Futami, and Hoshimiya 2001), which is reproduced below as Figure .

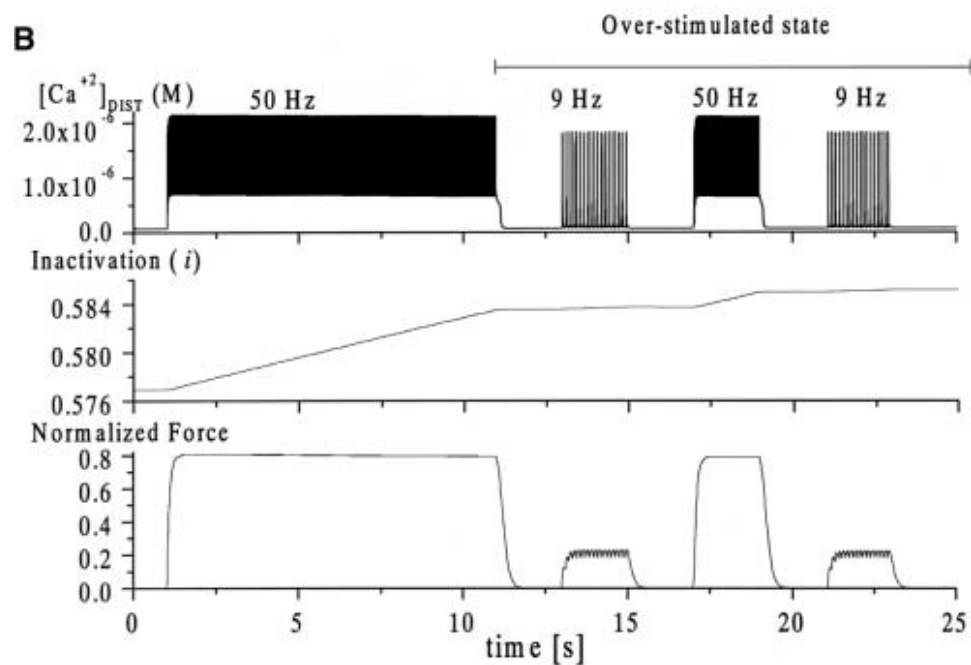


Figure 23. Figure 3B from (Otazu, Futami, and Hoshimiya 2001).

The results from the simulation using the HEMI M&S muscle model for distal calcium concentration are shown below in Figure .

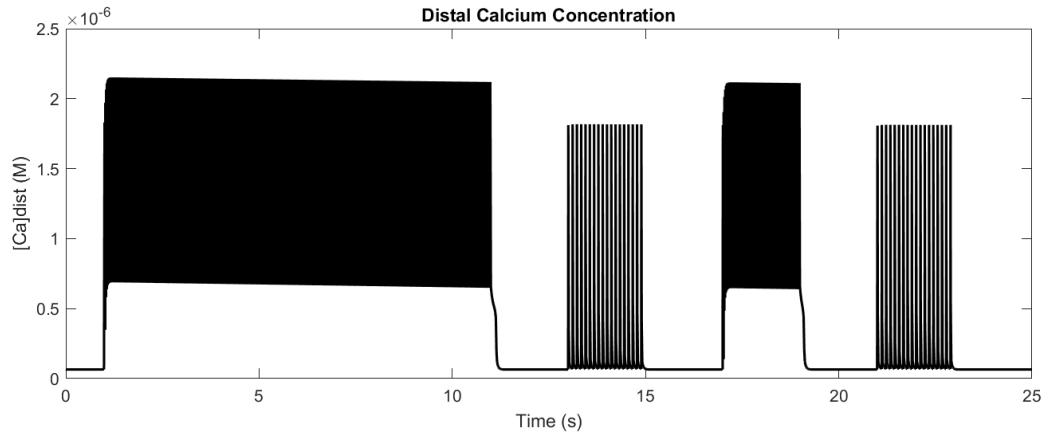


Figure 24. Distal calcium concentration results from the second verification test.

When compared to Figure , Figure provides verification that the Otazu model was properly implemented in the HEMI M&S system. Additional verification is obtained by comparing the fraction of bound inactivation sites, shown in Figure , to the values reported by Otazu et al. shown in Figure .

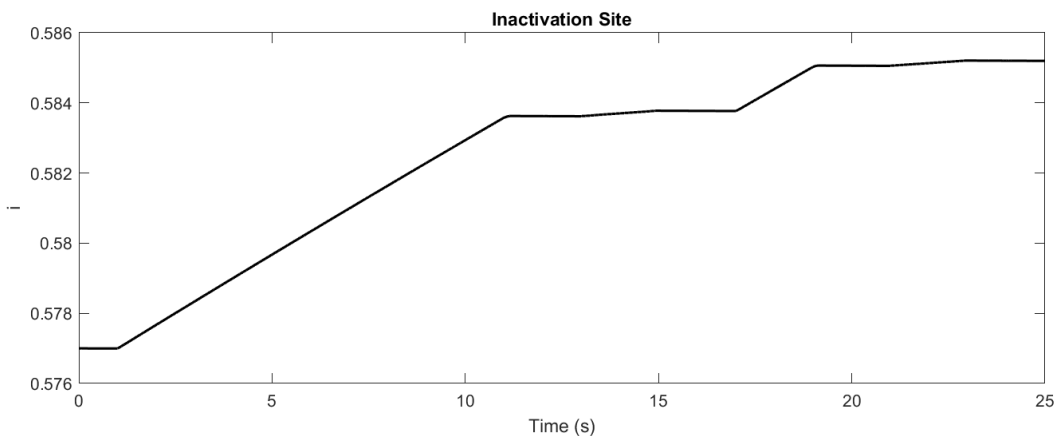


Figure 25. Inactivation site bound fraction results from the second verification test.

A.2.3. Muscle Verification Test 3

The third verification test compares the results of the simulation in Figure 6A in (Otazu, Futami, and Hoshimiya 2001) to the results produced by the HEMI M&S implementation. This simulation demonstrated the “catch-like effect,” where a single extra stimulation pulse causes a change in rates that leads to an increase in the mean calcium concentration and thus a greater fraction of activated troponin. the simulation begins with a 11.5 Hz stimulation at 0.2 seconds simulation time. After 0.6 seconds of simulation time, an extra pulse is added, and one second later a pulse is removed from the 11.5 Hz train. The results of this simulation are shown in Figure 6A in (Otazu, Futami, and Hoshimiya 2001), which is reproduced below as Figure .

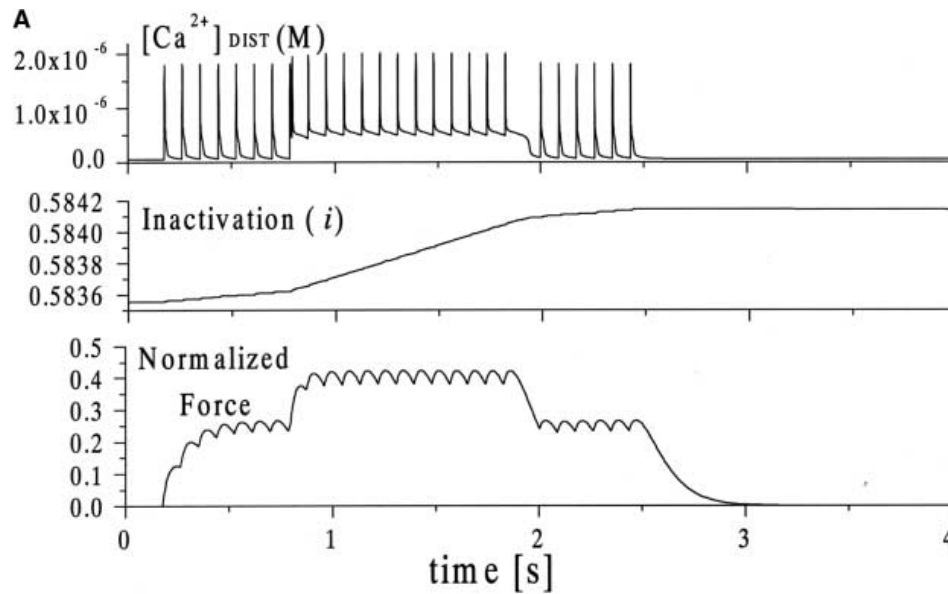


Figure 26. Figure 6A from (Otazu, Futami, and Hoshimiya 2001).

The distal calcium results from the simulation executed in the HEMI M&S system implementation are shown in Figure below.

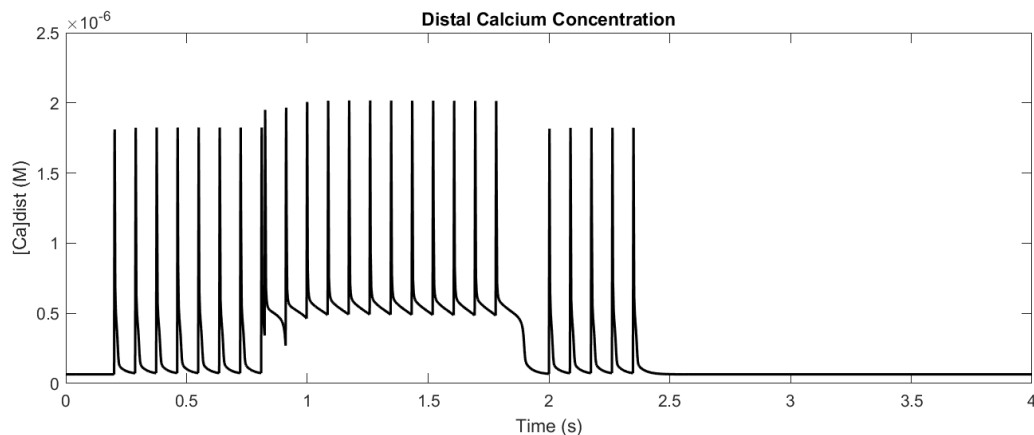


Figure 27. Distal calcium concentration results from the third verification test.

Verification of the implementation is achieved by comparing Figure to Figure . Additional verification comes from comparing the inactivation site bound fraction shown in Figure to Figure .

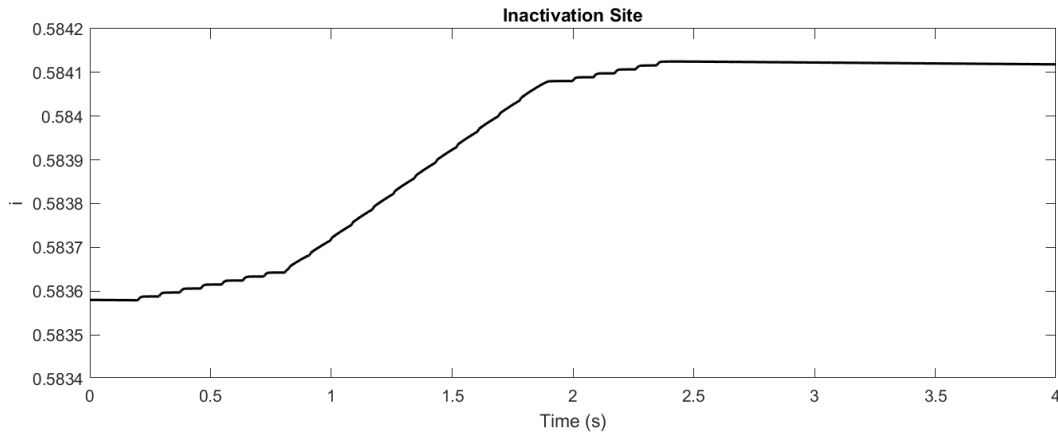


Figure 28. Inactivation site bound fraction results from the third verification test.

A.3. Conclusion

The implementation of the voltage model, the nerve model, and the muscle model have all been verified individually. Additionally, we have conducted initial integration tests that demonstrate that the composite model behaves as expected. For example, Figure is a plot of the distal calcium concentration during simulation where the HEMI M&S system is computing the body voltage, nerve activation, terminal transmembrane voltage, muscle activation, and muscle calcium dynamics.

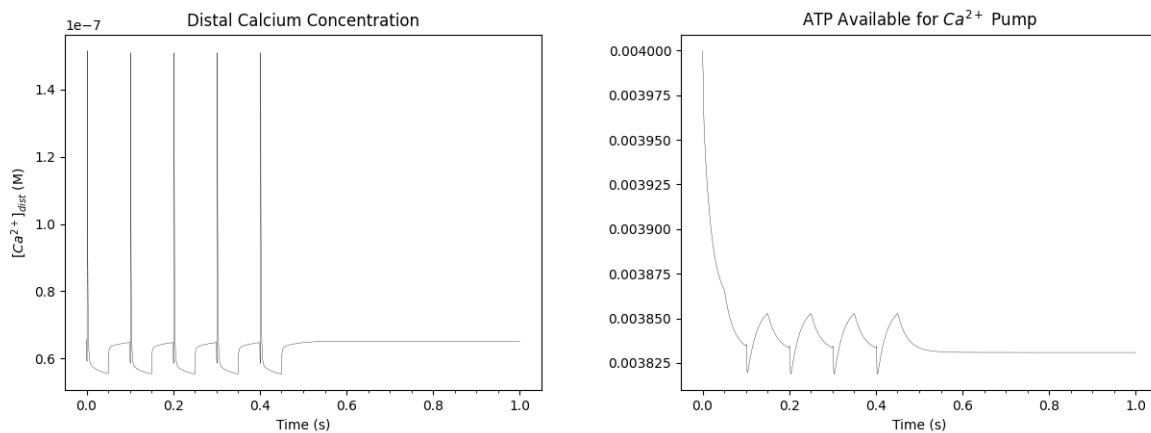


Figure 29. Distal calcium concentration (left) and pump-adjacent ATP concentration (right) during end-to-end simulation.

The results shown in Figure have not been validated, but they are reasonable and provide confidence that the integrated model can produce valid results.

The models that are being integrated into the HEMI M&S system to form the composite model were obtained from peer-reviewed literature, and the validity of these models has been established through the peer review process. In the next phase of this project, we intend to further establish model credibility through sensitivity analysis and uncertainty characterization. Additionally, we are working to identify data requirements for establishing the causality of events (see Section 7). These data may be applied toward validation of the computational model.

Appendix B. HEMI User Guide and Training Manual

An example of how to use the HEMI library is provided in the form of a Visual Studios project called `hemi-example` and which has only one source file called `main.cpp`. In the subsections to follow, we first describe how to build the HEMI library and its dependencies. We then describe how to build and execute a simulation by running the `hemi-example` executable. Afterwards, we examine the source in `main.cpp` to see how the simulation was built using the elements of the HEMI library.

B.1. Building the HEMI Library and its Dependencies

The first step for building the HEMI Library and its dependencies is to clone the HEMI repository. In the following, it is assumed the user has access to the repository via GitLab and that the user's machine has Git installed. To clone the repository, open the terminal prompt of your choice (for example MINGW64) and navigate to a preferred parent directory for your repository. For illustration, we will use `C:\Users\myusername\repos`. Then clone the repository as follows:

```
$ git clone https://git.ara.com/hemi/hemi.git
```

If this command is successful, there should now be a subdirectory of `C:\Users\myusername\repos` called `hemi`. Instructions for building HEMI's dependencies follow in the next section. We note that to build the HEMI dependencies and the HEMI library itself requires that CMake is installed on the user's machine.

B.2. Building the Dependencies

The HEMI library depends on a number of 3rd party libraries. They include, HDF5, Protocol Buffers, Boost, and GoogleTest. These must be available and locatable prior to any attempt to build the HEMI library itself. If these already exist on the user's system, the user can skip to section B.3

To build the HEMI library's dependencies, carry out the following steps:

1. Move to the directory `hemi\Dependencies`.
2. Create a subdirectory for the build. It is up to the user how to name this directory. For our illustration we will simply call it `build`.
3. Move into the `build` subdirectory, then execute the command:

```
$ cmake .. -DCMAKE_INSTALL_PREFIX=install
```

This will prepare the build system for the external dependencies and direct the results to be placed in the subdirectory `build\install`.

4. Next, execute:

```
$ cmake --build . --config=Debug
```

to build the Debug configuration of the dependencies. This step takes a while. Note that you can build a Release configuration instead by choosing `--config=Release`.

(**Note:** if `cmake --build . --config=Debug` gives you an error, try `cmake --build . --config Debug`. That is, replace the equal sign after `--config` with a space.)

Note that if you wish to be able to build both Debug and Release configurations of the HEMI library, it is necessary to build both the Debug and Release configurations of the HEMI dependencies.

B.3. Building the HEMI Library

Steps for building the HEMI library itself as well as the hemi-example sample project are as follows:

1. Move to the top directory of the repository. Using our illustration this would be `C:\Users\myusername\repos\hemi`. Once again create a build subdirectory in that directory with the name of your choice. As above for illustration, we will refer to this subdirectory as simply `build`.
2. Change into the newly created `build` subdirectory.
3. Execute the command:

```
$ cmake .. -DCMAKE_PREFIX_PATH='<absolute/path/to/Dependencies/install/directory>'
```

where `'<absolute/path/to/Dependencies/install/directory>'` is the absolute path to the install directory you created when you ran `cmake` for the external dependencies. Continuing with our illustration, this would be (note that the single quotes are needed!)
`'C:\Users\myusername\repos\hemi\Dependencies\build\install'`.

4. Open Visual Studio and choose “Open a Project or Solution.”
5. Navigate to the directory `C:\Users\myusername\repos\hemi\build` and choose `HEMI.sln`.
6. Select a configuration (e.g. Debug or Release) and then select `Build->Build Solution` (**Remember that only those build configurations for which the dependencies have also been built will result in a successful build.**)

B.4. Running hemi-example.exe

Once the steps in sections B.2 and B.3 have been successfully completed, then with the HEMI solution still open in Visual Studio, it should be possible to execute `hemi-example` by simply selecting `Debug->Start Without Debugging`. This will execute `hemi-example` with an empty argument list leading `hemi-example` to expect that the input file(s) required for the run will be found in `hemi\build`. As delivered, `hemi-example` expects to find three JSON formatted configuration files called `mynervein.json`, `myvoltagein.json`, and `mymusclein.json` respectively. These files are provided in a subdirectory of the `hemi` directory called `testhemi` and should be copied into `hemi\build` prior to running the example. Any output files generated during the run will also be deposited in `hemi\build` in this scenario.

Alternatively, the user may wish to run the hemi-example executable from the command line and provide it with an alternative directory to serve as the working directory for the simulation. At a minimum, before this can be done the user must supply the paths to the `dll` libraries for the dependencies. This can be done as follows:

1. In the search field of the windows task bar type “Environment Variables.” The option “Edit environment variables for your account” should appear. Select this option to open a dialog box.
2. The top half of the dialog box should show a window with a title similar to “User variables for myusername” with a list of variables provided in the window. One of these variables should be `Path`. Highlight the row containing this variable name and select Edit...
3. A second dialog box should now be open providing the list of paths which have been associated with the `Path` variable. Add a new path by selecting “new” and editing the new empty field that is added to the list as follows (where we continue with our above illustration):

```
C:\Users\myusername\repos\hemi\Dependencies\build\install\lib
```

4. Select Okay, then reopen the dialog box for editing the `Path` variable, and add a second new path by again selecting “New” and editing the new empty field added to the list as follows:

```
C:\Users\myusername\repos\hemi\Dependencies\build\install\bin
```

5. Select Okay again to close the `Path` editing dialog box, and then select Okay again to close the environment variables dialog box.
6. Close the terminal prompt instance that you have been using and open a new one (this will ensure that the new additions to the `Path` variable will be available.)

Once these steps have been taken choose or create a directory to be a working directory and place the required input files there. For illustration we select the directory `C:\Users\myusername\testhemi`. Once this directory is established and appropriately populated, carry out the following steps:

1. Navigate to the subdirectory containing the `hemi-example` executable. If, in our illustration, we assume that the `Release` configuration was built this will be the directory `C:\Users\myusername\hemi\build\HEMI\Release`. (Alternatively, you can add this path to the `Path` environment variable as done above for the Dependencies (remembering to close and reopen your terminal prompt afterward!) and then navigate instead to the `testhemi` subdirectory.)
2. Enter the command:

```
$ ./hemi-example C:\\Users\\myusername\\testhemi
```

3. Note the need to escape the backslashes in the path. To avoid the need for this, one may alternatively use single or double quotes around the path. For example,

```
$ ./hemi-example 'C:\Users\myusername\testhemi'
```

4. Note that if you have elected to add the directory containing the executable to your path you need only navigate to `testhemi` and simply enter the executable name with no argument as follows:

```
$ cd C:\Users\myusername\testhemi
```

```
$ hemi-example
```

The result should be a short HEMI simulation providing output files in `testhemi` that include a log file called `hemilogfile.log` containing some information about the run, a muscle data file called `m1_muscle_data_out.txt`, a JSON formatted file providing the parameters used for the muscle module called `mymuscleout.json`, a JSON formatted file providing the parameters used for the nerve module called `mynerveout.json`, a JSON formatted file providing the parameters used for the voltage module called `myvoltageout.json`, and a nerve data file containing a time series of the transmembrane voltages at the nodes of the nerve called `n1_tm_potentials_out.txt`.

B.5. Implementation of the hemi-example sample.

Finally, in this section we look at how the sample HEMI application `hemi-example` is constructed to understand how the modules are instantiated and made to work together to create a HEMI simulation. We need only focus on some of the contents of `main.cpp` to understand how the above sample is implemented.

We begin by noting that the first 100 or so lines are concerned with discerning what the working directory for the simulation is and do not concern the HEMI library itself at all. We therefore skip to the first line of interest which consists of a call to a function, defined in `Controller.h`, implemented in `Controller.cpp`, and called `CreateHemiController`, which returns a standard unique pointer to an instance of the `Controller` class. This line appears as follows:

```
std::unique_ptr<Controller> con = CreateHemiController(working_directory, logger);
```

Note that `logger`, is an object of type `Logger` which is instantiated on a previous line and which directs logging statements to be written to the file `hemilogfile.log` mentioned in section B.4.

Next, we instantiate a (Senn) nerve which we name “n1” as follows:

```
SESennNerve* n1 = DBG_NEW SESennNerve(“n1”, logger);
```

Here, `DBG_NEW` is a macro defined earlier in an `#ifdef` construct and with `_DEBUG` undefined, is simply a restating of the C++ `new` keyword. If `_DEBUG` is defined `DBG_NEW` is more complex, but either way, it serves to instantiate the object `n1`.

We next configure `n1` as follows:

```
n1->ConfigureFromFile("mynervein.json", working_directory)
```

At this point most of `n1` is initialized, the only exception being that the initial values for its internal vectors `y` and `dydt` have not yet been set. (This is done with a call to the `Initialize` method of `n1` which is done by the `Controller` itself).

Though not necessary, we can preserve the configuration of `n1` we've established via following call:

```
n1->SerializeToFile(working_directory + "\\mynerveout.json");
```

This is particularly useful for creating a template for JSON formatted files containing nerve parameters which can then be edited as desired by the user and used as alternative input files later. Similar method are available for the voltage and muscle modules as well.

Next, we add the nerve we've instantiated to the `Controller` object as follows:

```
con->AddSENerve(n1);
```

Now, it's time to choose a voltage module. In the present example we do this with the line:

```
SESennNativeVoltage* voltage = DBG_NEW SESennNativeVoltage(logger);
```

The we configure it:

```
voltage->ConfigureFromFile("myvoltagein.json", working_directory);
```

and add it to the controller:

```
con->AddVoltage(voltage);
```

We will also need a muscle object (which we name "`m1`") to pair with the nerve object:

```
SEWexlerMuscle* m1 = DBG_NEW SEWexlerMuscle("m1", logger);
```

Just as for the nerve and voltage objects, we may also configure it (this is not strictly necessary as default values for all muscle module parameters are supplied in the `SEWexlerMuscle` constructor):

```
m1->ConfigureFromFile("mymusclein.json", working_directory);
```

and add it to the `Controller` object:

```
con->AddSEMuscle(m1);
```

We can also ask `n1` and `m1` to output data from the simulation:

```
n1->SetRecordY(true);
```

```
m1->SetRecordY(true);
```

Before we can begin the simulation, we must remember to set the relevant global time variables (we choose units of milliseconds here):

```
con->SetGlobalTime(0.0, TimeUnit::ms);
con->SetGlobalSimulationDuration(4.0, TimeUnit::ms);
con->SetGlobalTimeIncrement(1.0e-03, TimeUnit::ms);
```

To start the simulation, we first call the Controller's Initialization method, and then execute:

```
con->Initialize();
con->execute();
```

The reader will note a few other lines not mentioned here, but those included here are sufficient to run the simulation discussed in section B.4.

Appendix C. Model Equations and Symbols

C.1. Neuromuscular Junction Model Parameters and Variables

For convenience, the neuromuscular junction model equation is repeated here, followed by a table of the model parameters and variables.

$$V_m(t) = V_n(t - \tau_{syn})$$

Table 5. Muscle model states

Variable	Description
t	time
V_n	presynaptic membrane potential
V_m	postsynaptic membrane potential
τ_{syn}	time delay in signal due to chemical transmission across synaptic cleft

C.2. Muscle Model Parameters and Variables

For convenience, the muscle model governing equations are repeated here, followed by tables of the model parameters and variables.

$$\begin{aligned} \frac{d[CaTN]}{dt} &= \alpha_{trop}[Ca^{2+}]_{dist}([TN] - [CaTN]) - \beta_{trop}[CaTN] \\ \frac{d[CaPARV]}{dt} &= \alpha_{parv}[Ca^{2+}]_{dist}([PARV] - [CaPARV]) - \beta_{parv}[CaPARV] \\ \frac{d[Ca^{2+}]_{prox}}{dt} &= \gamma_{volt} + \gamma_{Ca} + \gamma_{leak} - \gamma_{pump} - \frac{[Ca^{2+}]_{prox} - [Ca^{2+}]_{dist}}{\tau_{prox}} \end{aligned}$$

$$\begin{aligned}
\frac{d[Ca^{2+}]_{dist}}{dt} &= \frac{[Ca^{2+}]_{prox} - [Ca^{2+}]_{dist}}{\tau_{dist}} - \frac{d[CaTN]}{dt} - \frac{d[CaPARV]}{dt} \\
\frac{da}{dt} &= \alpha_a(1-a)([Ca^{2+}]_{prox})^2 - \beta_a a \\
\frac{di}{dt} &= \alpha_i(1-i)[Ca^{2+}]_{prox} - \beta_i i \\
\frac{d[ATP]}{dt} &= -\frac{1}{2}\gamma_{pump} + \kappa(\beta - [ATP]) \\
\frac{df}{dt} &= A[CaTN]\left(1 - \frac{f}{F_m}\right) - \frac{f}{\left(\tau_1 + \tau_2 \frac{[CaTN]}{[TN]}\right)} \\
\gamma_{Ca} &= f_{Ca}(1-i)a \\
\gamma_{pump} &= \frac{f_{pump}[Ca^{2+}]_{prox}}{K_{pump} + [Ca^{2+}]_{prox}} g([ATP]) \\
g([ATP]) &= (1 + e^{-k_1([ATP] - \tau_{ATP})})^{-1} \\
\gamma_{volt} &= \begin{cases} 0 & \text{no pulse} \\ f_{volt} \sin(250\pi t) & \text{pulse start time} < t < \text{pulse end time} \end{cases}
\end{aligned}$$

Table 6. Muscle model states

Variable	Description
[CaTN]	concentration of calcium that is bound to troponin
[CaPARV]	concentration of calcium that is bound to parvalbumin
[Ca ²⁺] _{prox}	calcium concentration in compartment proximal to sarcoplasmic reticulum
[Ca ²⁺] _{dist}	calcium concentration in compartment distal to sarcoplasmic reticulum
<i>a</i>	activation binding site bound concentration
<i>i</i>	inactivating binding site bound concentration
[ATP]	concentration of ATP available to the calcium pump
<i>f</i>	muscle force

Table 7. Muscle model state- and time-dependent rates

Variable	Description
γ_{Ca}	rate of calcium outflow from sarcoplasmic reticulum to proximal compartment
γ_{pump}	rate of pumped calcium outflow from proximal compartment to SR
γ_{volt}	rate of voltage activated calcium outflow from SR to proximal compartment

The muscle parameters are described in Table 8, and preliminary values for each parameter are also provided for convenience. Note that the formal parameter estimation process described previously has not yet been performed. Most of the values listed in the table come from literature and/or data, but they should be considered preliminary at the current stage of model

development, particularly the ATP production time constant (κ) and the ATP pump rate modification constant (k_1).

Table 8. Muscle model parameters

Parameter	Name	Value
[PARV]	total concentration of parvalbumin	$1.7 \times 10^{-4} M$
[TN]	total concentration of troponin	$9.9574 \times 10^{-5} M$
A	force development time constant	$1.45 \times 10^5 NM^{-1}s^{-1}$
f_{Ca}	maximum rate of the calcium activated channel	$0.52 s^{-1}$
f_{pump}	maximum pumping action	$1.3636 Ms^{-1}$
F_m	maximal theoretical force of muscle unit	$38.6 \times 10^{-3} N$
K_{pump}	calcium concentration at half maximum pumping rate	$5.1930 \times 10^{-6} M$
α_a	activating site binding constant	$7.2404 \times 10^{15} M^{-2}s^{-1}$
α_i	inactivating site binding constant	$1.8667 \times 10^3 M^{-1}s^{-1}$
α_{trop}	troponin binding constant	$1.8333 \times 10^7 M^{-1}s^{-1}$
α_{parv}	parvalbumin binding constant	$1.4 \times 10^7 M^{-1}s^{-1}$
β_a	activating site unbinding constant	$1.8975 \times 10^3 s^{-1}$
β_i	inactivating site unbinding constant	$9.3333 \times 10^{-5} s^{-1}$
β_{trop}	troponin unbinding constant	$600 s^{-1}$
β_{parv}	parvalbumin unbinding constant	$40 s^{-1}$
τ_{prox}	time constant for diffusion from proximal to distal compartment	$8.6445 \times 10^7 s^{-1}$
τ_{dist}	time constant for diffusion from distal to proximal compartment	$10\tau_{prox} s^{-1}$
τ_1	time constant of relaxation when [CaTN] is absent	$12.4 \times 10^{-3} s$
τ_2	time constant of force relaxation when [CaTN] is present	$274 \times 10^{-3} s$
γ_{leak}	Rate of calcium leakage from the SR to the proximal compartment	$0.0134 Ms^{-1}$
κ	ATP production time constant	$0.05 s^{-1}$
β	maximum available [ATP]	$4 \times 10^{-3} M$
k_1	ATP pump rate modification constant	$70,000 M^{-1}$
ξ	Probability of inhibition shape parameter	0.1
τ_{ATP}	[ATP] threshold below which calcium pump rate starts to decline	$4 \times 10^{-5} M$
f_{Ca}	maximum rate of $[Ca^{2+}]_{prox}$ that can be produced calcium activated channel	$0.52 Ms^{-1}$

Appendix D. FDTD Voltage

This appendix provides details of the FDTD Voltage implementation.

9.1.1.1.1. SEFDTDVoltage::EvaluateExternalPotentials

At the beginning of a simulation for which the SEFDTDVoltage module has been selected as a voltage module, `m_ept_ts` is empty. As part of its implementation for the SEFDTDVoltage

module, the public method `EvaluateExternalPotentials` calls a private method called `set_ept_ts` which fills `m_ept_ts` when `EvaluateExternalPotentials` is called for the first time during the simulation. The filling of `m_ept_ts` is a highly nontrivial process and most of `SEFDTDVoltage`'s private methods are written for the purpose of contributing to the accomplishment of this task. We delay a detailed discussion of how the filling of `m_ept_ts` is accomplished while completing a description of the implementation of `EvaluateExternalPotentials`.

Once `m_ept_ts` is filled during the first call, and for all subsequent calls to `EvaluateExternalPotentials`, the vector of external potentials required for a given time during the simulation is obtained by querying `m_ept_ts` with the time (It is recalled that `m_ept_ts` is a map whose keys are times). This is also a nontrivial task since `m_ept_ts` will, in general, not contain a value for the time used in the query. Instead, the querying time will likely fall between two time-values for which keys, and corresponding values, do exist in `m_ept_ts`. Hence, a vector of external potentials appropriate to the querying time must be constructed via interpolation using values of external potentials available at these nearby time-values. This process is encapsulated in another private method of `SEFDTDVoltage` called `get_ept` which takes a (double) time value as its sole argument and returns a vector of doubles providing the external potentials determined as just described. As for the case of `set_ept_ts`, we delay further description of `get_ept` while completing our description of `EvaluateExternalPotentials`.

Once `get_ept` has returned the required values of external potential, the final task of `EvaluateExternalPotentials` is to provide this information in a form that conforms to the requirements of the public interface of the method. We recall from our discussion of the `SEVoltage` base class that these potentials are to be placed in the container called `m_external_potentials` which is a standard ordered map whose keys are standard vectors with elements of type pointer to `SEPointType`, and whose values are of type pointer to `SEScalarElectricPotential`. Recall also that one of the arguments to `EvaluateExternalPotentials` is a reference to a standard vector with elements of type pointer to `SEPointType` which contain the coordinates of the locations where potentials are to be evaluated and which is referred to there as `coordinates`. To fill `m_external_potentials` therefore, a loop is constructed to loop over `coordinates` using an iterator of type `std::vector<SEPointType*>::iterator`. If `m_external_potentials` is empty (as it would be upon first call to `EvaluateExternalPotentials`), a new instantiation of type `SEScalarElectricPotential` is created and emplaced in `m_electric_potentials` at the location of the current value of the iterator. Then, whether newly emplaced or already emplaced, this quantity's `SetValue` method is called and used to set its value to the appropriate value from the vector returned by `get_ept`. Once this loop is complete, `EvaluateExternalPotentials` returns a reference to the now filled and/or updated container `m_electric_potentials`.

9.1.1.1.2. SEFDTDVoltage::set_ept_ts

We now return to a description of the method `set_ept_ts` which is called by `SEFDTDVoltage::EvaluateExternalPotentials` at the beginning of simulations employing the `SEFDTDVoltage` module and which fills the internal data member `m_ept_ts`. Recall that `m_ept_ts` is a map whose keys are times (coded as doubles) at which external potential values are presumed known or obtainable, and whose values are standard vectors of doubles whose values are the external potentials at the coordinates requested by the call to `set_ept_ts`. The task of filling of `m_ept_ts` is a matter of some complexity and requires the use of several complex data-structures as well as auxiliary tasks making use of the information contained in these structures and relegated to other private methods called by `set_ept_ts` in the course of its operation. To begin our description of the actions taken by `set_ept_ts`, it is best first to describe the data-structures mentioned above, after which we can provide an overview of the tasks carried out by `set_ept_ts`, and finally describe the actions of the auxiliary methods called by `set_ept_ts` whereupon our description of `set_ept_ts` will be complete.

9.1.1.1.3. Data Structures for set_ept_ts

At the beginning of a simulation, the only information available to the `SEFDTDVoltage` module is the name of the HDF5 file containing the mesh and electric field data, and some built-in assumptions about how that data is arranged within the file. Before `set_ept_ts` can evaluate external potentials, it must first have a means of holding information about the mesh and the electric field components in memory. Prior to opening and probing the HDF5 file, `SEFDTDVoltage` has no knowledge of the details of the mesh, or of how many datasets of electric field components are stored in the file. The data-storage information that `SEFDTDVoltage` *does have* at the outset of a simulation consists of the following:

- Mesh data can be found in the file in the “Mesh” group. This group consists of three simple 1-dimensional datasets containing the x, y, and z coordinates of the mesh. They are named “x”, “y”, and “z”, respectively, and their sizes can be ascertained using a member function of the HDF5 library.
- Electric field component datasets are found in the “/FieldData/TD” group. Each dataset has an attribute which provides the time for which the values of the field components are valid. Each dataset of field components is a simple 4-dimensional dataset of floating-point numbers. The first dimension is of size 3 and provides for distinguishing among the E_x , E_y , and E_z components of the electric field vector at a given mesh point. The sizes of the second, third, and fourth dimensions are the same as the sizes of the Mesh datasets “z”, “y”, and “x” respectively and thus serve to specify the z, y, and x coordinates of the mesh point to which the value of the field component is assigned.

These circumstances lead to the form for the data-structures adopted for containing mesh and electric field component data, as well as to the adoption of a third data-structure for containing the names and associated times of each of the field component datasets. We describe each of these data-structures in the following paragraphs.

For the mesh, we will find that a means must be ensured to allow for a search of the mesh for mesh points that are nearest neighbors to a given point within the volume. For this reason, the data-structure adopted for the mesh is a `struct` of type `mesh` defined to have 6 members and one member method called `fill_map`. (The actual member object so defined in `SEFDTDVoltage` is called `m_mesh`.) The data members of `struct mesh` consist of three standard maps `xmap`, `ymap`, and `zmap`, and three standard vectors `x`, `y`, and `z`. The standard vectors are of type `float` and contain the coordinates of the mesh along the Cartesian dimension after which the vectors are named. Thus, the vector `x` contains the mesh point `x` coordinates and so on. The maps have keys of type `float` which consist of the actual coordinate values for the Cartesian dimension after which the map is named and values of type `int` which provide the index to the corresponding vector where the coordinate can also be found. Thus, for example if the fourth element of the vector `x` is equal to 0.05, then we will have `x[3] = 0.05`, and the map `xmap` will have an entry with key equal to 0.05 and value equal to 3, or `xmap[0.05]=3`. As will be seen below, both of these storage types will be needed for the task of filling `m_ept_ts`.

The actual filling of the data members of `m_mesh` is done by `fill_map`. This method takes two arguments. The first argument provides an identification of the HDF5 file to be queried for mesh data. The second argument is an `enum` value of type `Mesh_Coords` with possible values of `X_COORD`, `Y_COORD`, and `Z_COORD`. (The `enum Mesh_Coords` is defined as part of the definition of the class `SEFDTDVoltage` itself.) `fill_map` is a low-level function which is called by a higher-level function called `validate_efield_time_series` which itself is called by `set_ept_ts`. We will discuss in greater detail how `fill_map` fills the members of `m_mesh` in the course of discussing `validate_efield_time_series` below.

The data-structure required for holding electric field component data is relatively simple by comparison. We note first that it will only be necessary to hold electric field component data from a single dataset at a time. This is because, as described below, we will have a means available for looping over the dataset names, permitting access to the data of each dataset in the HDF5 file one after the other. We have already noted that at the outset of a simulation the dimensionality of the datasets is known but the extent of each dimension (with the exception of the first) is unknown. We therefore define a data member of `SEFDTDVoltage` called `m_EFieldSnapshot` to be of type `float****`. The allocation of `m_EFieldSnapshot` is performed by a private member method of `SEFDTDVoltage` called `alloc_efield_snapshot`. This function is a low-level function called by `set_ept_ts`, and can only be called after the dimensional extents of the electric field component datasets have been ascertained. This will be discussed further below. (Note that `m_EFieldSnapshot` is only needed while `set_ept_ts` is filling `m_ept_ts`. It is deallocated via a call to the low-level function `dealloc_efield_snapshot` at the end of `set_ept_ts`.)

There is one more member data-structure of importance for the work done by `set_ept_ts`. This is in the form of a standard ordered map called `m_e_field_data_sets`. The keys for this map are of type standard `string` and represent the names of the datasets for the electric field components. The values for this map are defined to be `floats` and represent the times associated with the dataset specified by a given key. It will be recalled that the names of the datasets for the unaveraged electric field components are given by the corresponding OpenEMS

timestep for which the data are output. Recall too however, that a typical application of the `SEFDTDVoltage` module will have that module read from an HDF5 file containing time-averaged field datasets instead. As currently implemented, these are simply named in sequence with three-digit integers. Thus, for a file containing time-averaged data, the datasets will have the names, “000”, “001”, “002”, and so on. Additionally, the “time” attributes of the datasets provide the value of time for which the time-averaged fields are expected to be valid. These names and times are the names and times that are recorded in `m_e_field_data_sets`. The actual filling of this map with these keys and values is part of the work that is carried out by the member method `validate_efield_time_series` to be discussed below.

9.1.1.1.3.1. `set_ept_ts` Actions

With the data structures needed to facilitate the filling of `m_ept_ts` now described, we can proceed with the description of the actions of `set_ept_ts`. The first action taken by `set_ept_ts` is to call the method `validate_efield_time_series`. This function carries out the following work:

- Verifies that the file provided to `SEFDTDVoltage` exists and is a valid HDF5 file
- Determines the number of electric field component datasets the file contains.
- Loops over the electric field component datasets to acquire their names and associated times and uses this information to fill the map `m_e_field_data_sets`.
- Sets a data member `m_edims` to contain the extents of the dimensions of the datasets determined from the file. (This data member will be used later in `alloc_efield_snapshot` to specify the dimensional extents for the allocation of `m_EFieldSnapshot`.)
- Verifies that all of the datasets found are of the same dimensionality and extent.
- Calls the `fill_map` member method of `m_mesh` (discussed above) once for each Cartesian dimension to fill the data container members of `m_mesh`.
- Verifies that the sizes of the vector members of `m_mesh` are consistent with the extents of the associated dimensions of the electric field component datasets. For example, if `m_EFieldSnapshot` is to have extents `m_EFieldSnapshot[3][23][105][51]`, then `m_mesh.x.size()=51`, `m_mesh.y.size()=105`, and `m_mesh.z.size()=23`.

Once validation is complete, the next action taken by `set_ept_ts` is to allocate `m_EFieldSnapshot`. Recall that `validate_efield_time_series` preserves the dimensional extents found for the electric field component datasets in a 1-Dimensional array data member (of HDF5 data type `hsize_t`) called `m_edims`. The method `alloc_efield_snapshot` refers to this data member to determine the extents for the allocation of `m_EFieldSnapshot`. The process of allocation itself is straightforward and is readily understood from the code.

After the calls to `validate_efield_time_series` and `alloc_efield_snapshot`, `set_ept_ts` knows the locations of all mesh points in the volume, the names of all the datasets containing electric field information and the times with which they are associated, and

has space allocated for reading electric field data into memory so that potential values can be calculated. The next step is therefore to begin looping over the datasets and commencing the necessary calculations. Looping over datasets is facilitated by specifying an appropriately typed iterator to point to the beginning of `m_e_field_data_sets` and then commencing a `for`-loop on the iterator. Within this `for`-loop a number of complex operations are completed with (barring runtime errors) a filled `m_ept_ts` container as the outcome. Most of these operations are carried out by lower-level functions to be described below. We therefore summarize the operations of the main `set_ept_ts` `for`-loop as follows:

- The private method `read_efield_snapshot`, which takes a dataset name as its argument, is called to read a dataset from the HDF5 file and put its contents in `m_EFieldSnapshot`.
- The number of different locations (referred to below as “nodes”) where external potential values are needed is specified from the size of the vector `coordinates` in an integer variable called `nnodes`.
- Ordinary vector containers of type `double` are defined for holding `x`, `y`, and `z` coordinates of the nodes.
- An inner `for`-loop is established to range over the nodes at which external potentials are to be evaluated.
- Activity within the inner `for`-loop is as follows:
 - With integer `k` as the index for the `for`-loop, the Cartesian coordinates of the `k`’th node are obtained from `coordinates` and used to construct a variable of type `PointType` called `node_point`. (Note: `PointType` is a typedef for a standard tuple containing three floats. It is defined in `SEFDTDVoltage.h` and is not to be confused with the similarly named `SEPointType`. `PointType` is local and internal to `SEFDTDVoltage` while `SEPointType` is part of the Common Data Model. That these two types exist in the HEMI code is a historical accident.)
 - `node_point` is passed as an argument to the private method `find_nearest_mesh_points`. This method returns a vector of elements of type `PointIndexType`. This latter type is also a typedef defined in `SEFDTDVoltage.h` as a tuple of three ints providing the indices to the vectors `m_mesh.x`, `m_mesh.y`, and `m_mesh.z` respectively where the coordinates of a nearest neighbor mesh point can be found. The size of the vector returned by `find_nearest_mesh_points` depends on how many, if any, of `node_point`’s coordinates coincide with mesh point coordinates. If none of `node_point`’s coordinates coincide, `find_nearest_mesh_points` returns a vector of length 8. If one coordinate coincides, the vector returned is of length 4. For two coincident coordinates, the vector returned is of length 2, and if all coordinates coincide then `node_point` is itself a mesh point and only that point is returned in the vector. Details of how the vector returned by `find_nearest_mesh_points` is constructed are provided below.

- The vector `nearest_mesh_points`, returned by `find_nearest_mesh_points` is passed as an argument to the private member method `calculate_mesh_point_potentials`. This latter method returns a vector of floats called `mesh_potentials` which contain calculated values of the external potentials at the location(s) of the mesh points. How these calculations are done is the subject of a detailed description of `calculate_mesh_point_potentials` to be provided below.
- Next, `node_point`, `mesh_potentials`, and `nearest_mesh_points` are all passed as arguments to the private member method `interpolate_mesh_potentials_to_node` which carries out a linear, bilinear, or trilinear interpolation on the values of the mesh point potentials (depending on the lengths of the vectors containing mesh points and their potentials) to calculate a best value for the external potential at the node point itself. This value is returned as a float called `node_potential` which is subsequently added to a vector of doubles called `local_ept` via `static_cast`.
- At the end of the inner for-loop `local_ept` should contain `nnodes` values. This is verified using an `assert` statement after which `local_ept` is added as a value of `m_ept_ts` corresponding to a key equal to the time obtained from the attribute of the outer-loop's current dataset. `local_ept` is then cleared for reuse with the next dataset.

At the end of the outer for-loop `m_ept_ts` is filled with a number of sets of node potentials equal to the number of times for which electric field data sets are available in the HDF5 file. The final instruction contained in `set_ept_ts` is a call to `dealloc_efield_snapshot` which frees the memory associated with `m_EFieldSnapshot`. It remains to describe the function of the member functions called by `set_ept_ts`. We take these up in the subsections below.

9.1.1.1.3.2. SEFDTDVoltage::find_nearest_mesh_points

The first step in determining the value of an external potential at a node, is to find those mesh points that are nearest to the node and then to calculate potentials at those mesh points after which interpolation from the mesh point neighbors to the node is used to determine the potential at the node. The private member method `find_nearest_mesh_points`, called by `set_ept_ts`, is a function designed to accumulate a list of mesh points which are nearest neighbors to a node of interest. This function exploits the availability of the `find` and `lower_bound` methods which are part of the specification for C++ containers of type `std::map`. The process used by `find_nearest_mesh_points` to find coordinates of mesh points neighboring the node is similar for all three Cartesian coordinates of a node, so we describe it in detail here for only one of the dimensions.

Recall that the data member `m_mesh` has a member `xmap` which is a map whose keys are mesh x-coordinates and whose values are indices to vector `m_mesh.x` of x-coordinates. In other words, if `m_mesh.xmap` has key 0.05, and value 3, then `m_mesh.x[3]=0.05`. The method `find` of `xmap` returns an iterator, either to an element of `xmap` whose key is the argument of

`find`, or to `xmap.end()` if the key is not found. If the key is found, the coordinate is a mesh coordinate and each of two integers, `low_ix` and `high_ix` are set to the value contained in `xmap` whose key is the found key. If the key is not found, the coordinate is not a mesh coordinate and therefore presumably lies between two mesh coordinates. When this is the case, the method `lower_bound` of `xmap` is invoked on the coordinate. This returns an iterator to an element of `xmap` whose key is closest to but greater than the value of the coordinate. `high_ix` is set to this element's value. Subsequently the iterator is decremented once after which it now references the element of `xmap` whose key is closest to but less than the value of the coordinate. This element's value is set to `low_ix`. Hence, `low_ix` and `high_ix` now provide indices for `m_mesh.x` which give the closest mesh x-coordinates to the node's x-coordinate.

The procedure above is repeated above for the y and z coordinates of the node so that in the end, 3 pairs of high and low indices are collected which point to the mesh coordinates which are either equal to, or closest to but either greater or less than the node coordinates along each Cartesian dimension. Once these are determined, a standard vector of type `PointIndexType` (see above discussion) is declared to contain the mesh points which neighbor the node. Filling this vector properly requires an examination of the values of the index pairs to determine which and how many neighbors the node actually has. For example, if all three pairs exhibit equality, then all three node coordinates are mesh coordinates which means that the node is a mesh point. When this occurs, the only mesh point at which an electrical potential must be calculated is the node itself. Hence `find_nearest_mesh_points` constructs a single instance of type `PointIndexType` whose indices are the indices to the mesh point whose coordinates are the node's coordinates and adds that single entry to the vector it returns.

At the other extreme, it can happen that none of the node coordinates are in common with mesh point coordinates in which case all indices collected will be pairwise unequal. In this case the node point is a fully interior point and will have a total of eight neighbors each of whose coordinate indices will be comprised of one of eight possible combinations of those indices. Hence eight tuples of type `PointIndexType` are constructed and added to the vector returned by `find_nearest_mesh_points`. In between these two extremes are several other possible cases all of which are handled by a somewhat elaborate set of conditionals testing for pairwise equality of the index pairs along each dimension. All possibilities are thus accounted for and the length and contents of the returned vector of mesh points is determined by which possibility is realized for each node.

9.1.1.1.3.3. SEFDTDVoltage::calculate_mesh_point_potentials

For each node at which an external potential is required, a set of neighboring mesh points is collected by the function `find_nearest_mesh_points` discussed above. Once these are available the next step in determining the node potential is to calculate the potentials at the mesh points. This is the task of `calculate_mesh_point_potentials` which is called by the function `set_ept_ts` as described above. This function takes the vector of neighboring mesh points as an argument, and loops over it, calculating potentials for each mesh point and emplacing the results as elements of a vector of type `double` after which the vector is returned to `set_ept_ts`.

The calculation of a mesh point potential is the same for all mesh points and thus we need only describe the calculation of one such mesh point. We recall first that the electrostatic potential difference between two points p_1 and p_2 in a volume containing an electric field $\vec{E}$ is given by the line integral shown in Equation (19), where $d\vec{l}$ is a line element along the path of integration.

$$\Delta U = - \int_{p_1}^{p_2} \vec{E} \cdot d\vec{l} \quad (19)$$

One of these two points (p_1) is the mesh point itself whereas the other is a point of reference (common to all mesh points) the appropriate choice for which is the mesh point corresponding to the presumed location of the anode of the HEMI weapon being deployed to create the electric field. (It is assumed that the external potentials sought are the values of voltage one would obtain from measurement via voltmeter when placing one of the voltmeter's probes at the location of this anode and the other at the location of the mesh point (or node point). The coordinates of the reference point must be specified by the user conducting the simulation by specifying the values of the three input parameters `AnodeXCoordinate`, `AnodeYCoordinate`, and `AnodeZCoordinate`. Typically, these are set in a JSON formatted configuration file which is read in as part of the process of instantiating an `SEFDTDVoltage` object. They are added to the Common Data Model and made available to `calculate_mesh_potentials` so that this function can construct a local variable called `anode_point` of type `PointType` (see above) which then serves as the reference point for the line integrations to follow.

After `anode_point` has been constructed and prior to line integration, some checks on the value of `anode_point` are made to ensure that line integration can proceed. In particular, the coordinates of `anode_point` are compared to the upper and lower limits on coordinates of the mesh along each dimension to ensure that the reference point lies within the volume for which electric field data is available. Additionally, the function `find_nearest_mesh_points` is applied to `anode_point` as a convenient means of confirming that `anode_point` is in fact a mesh point. (A current limitation on `SEFDTDVoltage` is that `anode_point` must be a mesh point.) If `find_nearest_mesh_points` returns a vector containing only 1 element, then `anode_point` is a mesh point. If either of these two conditions on `anode_point` fails, an error is logged, and the simulation halts.

Calculation of the line integral between a mesh point and reference point is a matter of selecting a path along which to integrate and applying a numerical approximation to the equation above. Since we cannot predict in advance which points will be integrated between, we are limited in our choice of numerical method for integration. The method chosen for `SEFDTDVoltage` is the trapezoid rule. (Other more sophisticated methods require constraints on the number, oddness and/or evenness of the points along the path of integration.) As an example, let $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_{N_x}, y_{N_y}, z_{N_z})$, where N_x is the number of mesh points lying along the x direction and inclusive of x_1 and N_x (and similarly for N_y and N_z). Finally let $E_{x;i,j,k}$ be the x-component of the electric field at the point $p = (x_i, y_j, z_k)$ (and similarly for $E_{y;i,j,k}$ and $E_{z;i,j,k}$). Then, if we choose a path from p_1 to p_2 such as to move along the x direction first, the y direction second,

and the z direction third according to $p_1 \rightarrow (x_{N_x}, y_1, z_1) \rightarrow (x_{N_x}, y_{N_y}, z_1) \rightarrow p_2$, then our numerical integration scheme is given by Equation (20).

$$\begin{aligned} \Delta U \approx & - \sum_{i=1}^{N_x-1} (x_{i+1} - x_i) (E_{x;i+1,1,1} - E_{x;i,1,1})/2 \\ & - \sum_{j=1}^{N_y-1} (y_{j+1} - y_j) (E_{y;N_x,j+1,1} - E_{y;N_x,j,1})/2 \\ & - \sum_{k=1}^{N_z-1} (z_{k+1} - z_k) (E_{z;N_x,N_y,k+1} - E_{z;N_x,N_y,k})/2 \end{aligned} \quad (20)$$

Different paths are, of course, possible. For example, we could move along x first as before, but then move along z prior to moving along y. Then our scheme would be given by Equation (21).

$$\begin{aligned} \Delta U \approx & - \sum_{i=1}^{N_x-1} (x_{i+1} - x_i) (E_{x;i+1,1,1} - E_{x;i,1,1})/2 \\ & - \sum_{j=1}^{N_y-1} (y_{j+1} - y_j) (E_{y;N_x,j+1,N_z} - E_{y;N_x,j,N_z})/2 \\ & - \sum_{k=1}^{N_z-1} (z_{k+1} - z_k) (E_{z;N_x,1,k+1} - E_{z;N_x,1,k})/2 \end{aligned} \quad (21)$$

Schemes for 6 possible path orders have been implemented in SEFDTDVoltage. They are found in the private member function `calculate_potential_from_line_integral` which takes a mesh point, a reference point and an enum value of type `Line_Int_Mode` as arguments. The enum `Line_Int_Mode` is defined in `SEFDTDVoltage.h` as part of the `SEFDTDVoltage` class and has six possible values named:

`INT_XYZ`, `INT_XZY`, `INT_YXZ`, `INT_YZX`, `INT_ZXY`, and `INT_ZYX`.

When `calculate_potential_from_line_integral` is called, a switch statement tests for the enum value that was passed to it and selects the path of integration accordingly.

The need for multiple paths of integration arises because, as pointed out earlier, external potential values are only well-defined when the field used to calculate them is quasi-static. For a field that is quasi-static, the line integral given above will provide values that are the same, or at least nearly so, for any choice of path connecting the endpoints of integration. We have discussed at length above the need for time-averaging the electric field data provided by the electrical dosimetry module before use in the `SEFDTDVoltage` module in order to help ensure that the field data used in this module are indeed quasi-static. Time-averaging alone however does not guarantee that the resulting averaged data will be quasi-static and thus it is necessary to

test for this condition by carrying out multiple integrations along differing paths and testing that all integration paths lead to candidate values for a mesh potential which are approximately equal to them within an acceptable tolerance. With all of the foregoing understood, the actions of `calculate_mesh_point_potentials` are as follows:

- `anode_point` is initialized according to the values input by the user for the parameters `AnodeXCoordinate`, `AnodeYCoordinate`, and `AnodeZCoordinate` as discussed above. It is subsequently tested to ensure that it lies within the mesh volume and is itself a mesh point.
- A `for`-loop is established which loops over the elements of the vector of mesh points which has been passed as an argument to the method. Within this `for`-loop the following steps are taken:
 - For each mesh point, six “candidate” values for the value of the mesh point potential are calculated by successive calls to `calculate_potential_from_line_integral`. For all six calls to this function, the same mesh point and reference point (i.e. `anode_point`) are given as arguments but the path of integration to use is changed each time. Each value is stored separately in a vector called `candidate_potentials`.
 - Candidate potentials are averaged with the result placed in `comparison_potential`.
 - For a non-zero `comparison_potential`, a `for`-loop over the candidate potentials is initiated within which a fractional difference between each candidate potential and the comparison potential is calculated and checked against a tolerance called `POT_TOLERANCE` which is defined and specified in `SEFDTDVoltage.h`. If the fractional difference falls within the tolerance the loop is continued, otherwise a path independence failure warning is issued, and the simulation is halted.
 - For near-zero values of `comparison_potential` the individual values of `candidate_potentials` are examined to confirm that they are all separately near zero to within the same limit as was used for testing the nearness to zero of the value of `comparison_potential`. If so, the loop continues. Otherwise, the simulation halts with a path-independence failure warning as above.
 - If the potential calculated for the mesh point passes the path-independence test, it is added to a vector of results called `result` and the `for`-loop continues on to the next mesh point.
- Upon successful completion of the `for`-loop over mesh points, the `result` vector is returned to calling function (`set_ept_ts`).

In tests run using time-averaged electric field data obtained from averaging 200 unaveraged datasets from within a 4-microsecond time interval to obtain time-averaged datasets separated in time by a time interval of approximately 2 microseconds each, we were able to successfully test for path-independence for a tolerance of 5% over 44 datasets and all mesh point neighbors for

51 separate nodes. (We note however that occasional outlying results do occur depending on the choice made for anode coordinates.)

9.1.1.1.4. SEFDTDVoltage::get_ept

In our description of `EvaluateExternalPotentials` it was mentioned that the private member `get_ept` was responsible for querying `m_ept_ts` for external potential values at node coordinates for a time requested by the call to `EvaluateExternalPotentials`, and that this querying process was non-trivial because the querying times are likely not included among keys of the map `m_ept_ts` containing external potential data. There are two reasons why the querying time will in general not be equal to any of the keys (which represent times for which the external potentials are known) available in `m_ept_ts`.

Firstly, the global time increment set for a HEMI simulation is independent of the time increment selected for the OpenEMS electrical dosimetry script. Indeed, this latter time step cannot be directly chosen by the user of the OpenEMS script as its value is constrained by the mesh resolution chosen and the limits on the value of the Courant number for the Finite Difference Time Domain method. Typically, OpenEMS time increments are several orders of magnitude smaller than those required by the integrators used by the other HEMI modules, but the specific value is calculated internally and merely reported by the script. So, using the same example for OpenEMS timestep as used in previous sections, (that is, $\Delta t_{\text{EMS}} = 1.92583 \times 10^{-11} \text{ s}$) and supposing we have set our HEMI time increment to $\Delta t_{\text{HEMI}} = 2.0 \times 10^{-6} \text{ s}$ we find that the number of OpenEMS timesteps spanning two HEMI timesteps is, $\text{ceil}(2 \Delta t_{\text{HEMI}} / \Delta t_{\text{EMS}}) = 207704$. But the actual time spanned by this number of OpenEMS time steps is $207,704 \times 1.92583 \times 10^{-11} \text{ s} = 4.000025943 \times 10^{-6} \text{ s}$ which is the value of the time that would be output as an attribute to a dataset reported after this number of timesteps. The issue is compounded by the fact that an average of such datasets must be taken over the range $2 \Delta t_{\text{HEMI}}$ to determine a time-averaged field to be reported for a time lying roughly half-way between the beginning and ending times of the range over which the averaging has been done. This time, which is simply the average of the times within the range for which unaveraged datasets are available, cannot be constrained in such a way as to provide a time increment between time-averaged field datasets which is precisely equal to the HEMI timestep for the simulation.

A second reason why `m_ept_ts` cannot be expected to contain information for the precise querying times for which `get_ept` is likely to be called has to do with the numerical integration method employed by the nerve module. Recall that `get_ept` is called by `EvaluateExternalPotentials` which is called by a nerve module in the course of that module's process of carrying out its own numerical integration. The methods used by a nerve module to carry out its own integration are likely to require information at times that lie between the times spanned by a HEMI time increment. For example, the module `SESennNerve` uses a 4th-order Runge-Kutta method which requires values of external potential at several times lying between the current time and the time advanced by a global HEMI timestep. Hence for both of the reasons presented, it has been necessary to anticipate the need for interpolation between times for which external potentials are known. With this understood we can describe the actions of `get_ept` as follows:

- Upon entry, a vector of doubles called `ept` is declared to contain the results to be passed back to the calling function (`EvaluateExternalPotentials`).
- A suitably typed iterator is then declared and then set to point to the last entry of `m_ept_ts`. Also, the value of the HEMI simulation's pulse duration is copied to a local variable `xpd`.
- If the time passed to `get_ept` is greater than the duration of the pulse, then the vector of results is populated with zero values for all of the coordinates for which external potentials are required and `get_ept` returns the vector.
- If the time passed to `get_ept` is less than or equal to the duration of the pulse, then the following steps are taken:
 - A second local time variable `tt` is declared. If the time passed to `get_ept` is less than or equal to the time corresponding to the key value obtained from the last entry in `m_ept_ts`, then `tt` is set equal to this time. Otherwise, `tt` is set to a real-valued “modulus” of the current time. That is, the number of times the current time can be divided by the key for the last entry of `m_ept_ts` is determined. Then the time equal to this number multiplied by the last key value is determined and subtracted from the current time. This provides a time that will lie within the range of times for which keys exist in `m_ept_ts`. (Recall the severe limitations on the number of time-averaged datasets available due to runtime restrictions mentioned earlier, hence the need to “cycle” through available time entries).
 - Once `tt` is determined, the `find` method of `m_ept_ts` is used to reset the iterator to elements of `m_ept_ts` declared previously to point either to `m_ept_ts.end()` or to an element of `m_ept_ts` with key equal to `tt`. In the latter (and very unlikely) case, the external potentials required are contained in the vector value of the element of `m_ept_ts` to which the iterator points. They are thus assigned to `ept` and returned to the calling environment.
 - If (as is much more likely) the iterator points to `m_ept_ts.end()` the following steps are taken:
 - Two vectors of doubles, `ept_0` and `ept_1`, are declared.
 - The `lower_bound` method of `m_ept_ts` is used to find an iterator that points to an element `m_ept_ts` whose key is the smallest key greater than the value of `tt`.
 - A time variable `t_1` is set to the key of element found and `ept_1` is set to the corresponding value. The iterator is decremented and a second time `t_0` is set to the key of the element to which it now points. Subsequently, `ept_0` is set to the value of this element.
 - If the difference between `tt` and `t_1` is negligible, `ept` is set to `ept_1`. Alternatively, if the difference between `tt` and `t_0` is negligible, `ept` is set to `ept_0`. If neither difference is negligible, `ept` is evaluated via

linear interpolation using the values contained in `ept_0` and `ept_1` as well as the values of the times `t_0` and `t_1`.

- Optionally (depending on the value of a boolean member set by the user called `m_record_vext`), after `ept` has been determined via the above steps, `get_ept` records the time and the external potential values in a file.