POLITECNICO DI TORINO

Master's Degree
in Computer Engineering

Master Thesis

A Python framework for the development of hybrid models of neuromodulation

Supervisors:
prof. Silvestro Micera
prof. Gabriella Olmo

Author:
Claudia Sabatini

Academic Year 2022 - 2023

Un altro strappo lungo i bordi

Summary

Computational models provide mathematical abstractions of real-world problems and are typically used to understand the behavior of a physical system, formulate hypotheses, and make predictions, while reducing the economical and ethical cost of experiments. In the field of neuromodulation, modeling the effects of neural stimulation is a fundamental step in the design of neuroprosthetic devices. Currently, the so-called hybrid models (HMs), which encompass the problems of volume conduction and neural response computation, have been successfully employed in the context of spinal cord stimulation, deep brain stimulation and peripheral nerve stimulation. At the state of the art, the main weakness of HMs is their high computational cost and the consequent difficulty of parameter optimizations requiring many model evaluations. To partially overcome these limitations, we can resort to the use of surrogate models, which leverage machine learning techniques to predict simulation outcomes abstracting from biophysical details. Moreover, the modularity of the HM framework allows to reuse a handful of fundamental tools for very different neuroprosthetic applications. For these reasons, we propose to build a framework to build HMs of peripheral nerve and spinal cord stimulation using Python, an object-oriented programming language (OOP), widely used for machine learning. Our models range from simplified models to very complex models of spinal cord stimulation including vertebral geometries and elaborate fascicular morphologies in peripheral nerves. Starting with the translation in Python of HMs of nerves with a relatively simple morphology already present in the literature, we have defined a method that allows for the automatic and programmatic generation of nerves models with merging, splitting and rotating fascicles. We show how to train a surrogate model, based on 3D UNet, a convolutional neural network architecture mainly employed for biomedical image segmentation, in the cases of nerves whose fascicles follow straight or curved paths. We show that the building blocks used to create a model are the same, regardless of the geometry to be created. Furthermore, the extensive use of OOP language paves the way for the creation of a single software suite capable of generating any model with simple integrations, such as the addition of specific classes, to the existing codebase. Our framework has been thoroughly documented and use cases and tutorials have been provided, to maximize the usability and expandability of the framework, both through the inclusion of further modelling modules and machine learning surrogate models

Acknowledgements

I would like to express my sincere gratitude to my supervisors for their support and guidance throughout this thesis. Thank you Prof. Micera for providing me with the opportunity to work in your laboratory and for your belief in my potential. Thank you Prof. Olmo for your availability and valuable advice.

Contents

List of Tables

List of Figures

Part I

Chapter1

Chapter 1

Introduction

In translational studies of neurostimulation, computer models have proven to be highly effective tools for research and experimentation. Computational models for neurostimulation are built upon the principles introduced by Hodgkin and Huxley in 1952 [19].
These clinically oriented applications originated from the necessity to comprehend the mechanisms of the electrical stimulation of axons in restoring sensory-motor functions in disabled subjects [12].
Computational tools are extensively used in many contexts, such as peripheral nerve stimulation, spinal cord stimulation and deep brain stimulation.

This thesis aims to build a user-friendly framework for designing spinal cord and peripheral nerves stimulation models and to investigate the use of machine learning-based models for optimizing traditional ones.

1.1 Overview Nervous System

The nervous system is a complex network of specialized cells and tissues that play a crucial role in controlling and coordinating various functions in the human body.
The nervous system is essential for life, it allows us to interact with our environment, control our bodies, and experience the world around us. Damage to the nervous system can lead to a variety of problems, including paralysis, sensory loss, cognitive impairment, and seizures.
The vertebrate nervous system is anatomically divided into two main parts: the central nervous system (CNS) and the peripheral nervous system (PNS) [24].
The CNS consists of the brain (cerebral hemispheres, diencephalon, cerebellum, and brainstem) and the spinal cord. While the PNS includes all the nerves and ganglia (collections of nerve cell bodies) outside the CNS [24].

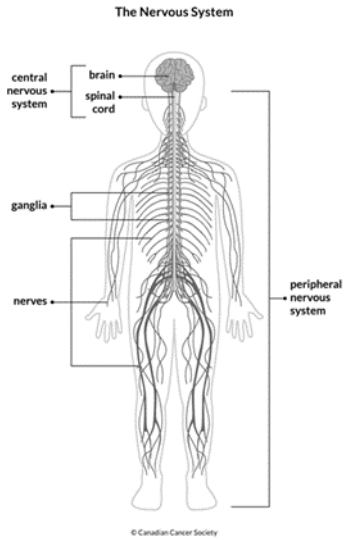


Figure 1.1. Overview Nervous System

1.2 Neurons

The nervous system operates through the communication of specialized cells called neurons that conveys electrochemical impulses throughout the body.
Neurons consist of a central body which contains the nucleus that houses the genetic material (DNA) and plays a vital role in maintaining the cell's metabolic processes, an axon and the elaborate arborization of dendrites which are the most obvious morphological sign of neuronal specialization for communication [24].
The axon is a long, slender, and typically unique unbranched projection that extends from the cell body. It may travel a few hundred micrometers or much farther, depending on the type of neuron [24], for example, the axons that run from the human spinal cord to the foot are about a meter long while the axons in the nerve cells in the human brain are about no more than a few millimeters long. Axons convey electrical signals over such distances by a self-regenerating wave of electrical activity called an action potential or "spike" [24].
Moreover, in some neurons, the axon is wrapped in a fatty myelin sheath, which is produced by cells called oligodendrocytes in the CNS and Schwann cells in the PNS. In myelinated axons, the morphology includes distinct regions known as the node of Ranvier (NoR), internode (IN), and paranode (PN). These regions play critical roles in facilitating the rapid and efficient conduction of nerve impulses along the axon. In particular NoR is a gap in the myelin sheath where the axon membrane is exposed and capable of transmitting electrical signals. The IN is the myelinated segment of the axon between nodes, and the PN is the region adjacent to the node responsible for maintaining the myelin sheath's integrity and controlling ion flow.

At the end of axon, there are small structures called synaptic terminals which form synapses, used by neurons to communicate with each other.
Dendrites are branching extensions that extend from the cell body. They receive incoming signals, usually in the form of neurotransmitter molecules released by other neurons. Dendrites have numerous small protrusions called dendritic spines, where many synapses with other neurons occur.
The variation in the size and branching of dendrites is enormous, and of critical importance in establishing the information-processing capacity of individual neurons in fact neurons with more dendrites are able to receive and integrate information from a larger number of other neurons.
Neurons can be classified based on the number of dendrites they have, they can be unipolar, bipolar, multipolar and pseudounipolar.

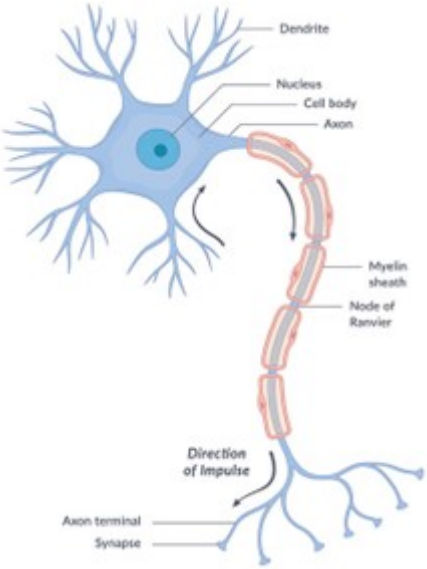


Figure 1.2. Structure of a neuron.

1.3 Flow of information between CNS and periphery

This flow of information between the CNS and the periphery is fundamental to how the nervous system functions.
It allows us to perceive and respond to our environment, control our movements, and maintain various bodily functions.
The flow of information between CNS and the rest of the body involves a complex process that includes both sensory and motor pathways.
Sensory neurons, also known as afferent neurons, are responsible for transmitting information from sensory receptors in the periphery to the CNS. The sensory neurons relay the

information to the spinal cord or directly to the brain, depending on the type and location of the sensory receptor. These specialized cells or structures are located in the skin, muscles, organs, or sensory organs and detect various stimuli, such as touch, temperature, pain, and sensory information from the special senses (vision, hearing, taste, smell).
When a sensory receptor detects a stimulus, it generates electrical signals known as action potentials. In some cases, the information is first processed at the spinal cord level, while in others, it is transmitted to higher brain centers for conscious perception. Within the CNS, the information is processed and integrated. This may involve the perception of sensory stimuli, the coordination of motor responses, or the initiation of reflex actions.
Motor neurons, also known as efferent neurons, are responsible for transmitting commands and motor signals from the CNS to muscles and glands in the periphery.
There are two main types of motor neurons: upper motor neurons, which, are located in the CNS, typically in the brain or spinal cord, and provide input to lower motor neurons, and lower motor neurons, which are located in the spinal cord or in the brainstem, and they directly innervate muscles or glands. The axons of lower motor neurons extend to the muscles or glands they control. When stimulated by motor signals, muscles contract, and glands secrete substances.
Motor pathways also involve feedback mechanisms, such as proprioceptive information from muscles and tendons, to regulate and adjust motor responses.
The CNS continually integrates sensory information, processes it, and generates motor responses. The feedback loop allows the CNS to adjust motor commands based on sensory input and maintain homeostasis.

The autonomic nervous system is responsible for controlling the body's involuntary functions, such as heart rate, breathing, blood pressure, and digestion. The autonomic nervous system can be either afferent (sensory) or efferent (motor).
There are also primary neurons which are neurons that are in direct contact with the periphery. They can be either afferent or efferent. For example, sensory neurons in the skin are primary afferent neurons, while motor neurons in the muscles are primary efferent neurons.
Interneurons are neurons that are located in the central nervous system. They receive and send information to other neurons. Interneurons can be involved in a variety of functions, such as processing sensory information, controlling movement, and regulating emotions.
Here an example on how the communication works: afferent neurons in the skin detect pain and send information to the central nervous system. Interneurons in the spinal cord process this information and send signals to the brain and to other motor neurons. The motor neurons then send signals to the muscles in the arm to cause the arm to be withdrawn from the source of pain.

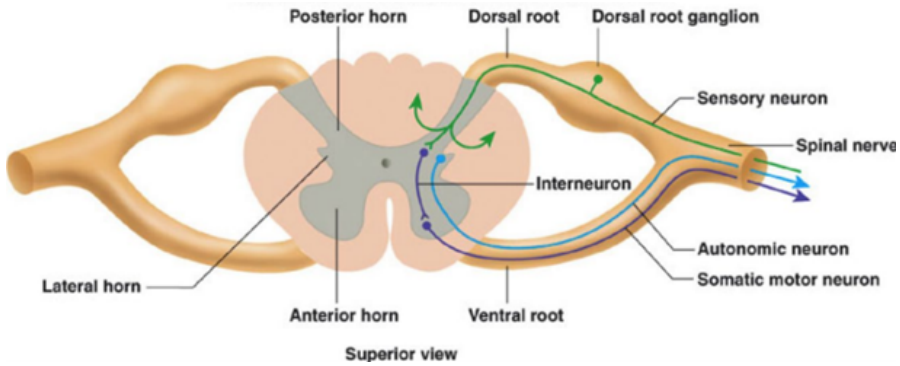


Figure 1.3. Flow of information between CNS and periphery.

1.4 Nerve and Nerve Fibers

Peripheral axons are gathered into bundles called nerves. These axons are usually referred to as nerve fibers. Nerve fibers can be classified based on various criteria, including their size, function, and the presence or absence of myelin sheaths.
Based on size, they can be classified into Aα, Aβ, Aδ and C. Aα fibers are large-caliber (diameter 12–20 µm) myelinated fibers, Aβ fibers are smaller (diameter 6–12 µm) myelinated fibers, Aδ fibers are small (diameter 1–5 µm) slightly myelinated fibers and C fibers are very small (diameter 0.2–2 µm) unmyelinated fibers [26].
Nerve fibers are organized into macroscopic structures called nerve fascicles. The nerve fibers in a fascicle are bathed in a spongy cushion of connective tissue called the endoneurium [26], which is mostly made up of collagen, helping to hold the nerve fibers and blood vessels in place.
Fascicles are enclosed by a protective connective tissue called the epineurium. The epineurium is mainly made up of collagen fibers and contains fat cells called adipocytes. Moreover, the epineurium also provides structural support for tiny blood vessels known as vasa nervorum. These blood vessels extend from the epineurium and form a network of small capillaries that penetrate into the endoneurium. So, the epineurium not only protects the nerve fascicles but also facilitates the delivery of blood and nutrients to the individual nerve fibers through these tiny blood vessels.
Fascicles are separated from the surrounding epineurial tissue by a thin layer of connective tissue called perineurium. Perineurial cells are tightly connected to each other through tight junctions. These tight junctions serve a crucial function by reducing electrical conductivity within the perineurial sheath. This reduction in electrical conductivity helps create a barrier that can protect nerve fibers from injury, both mechanically and chemically.
In larger nerves, fascicles are grouped together into bundles that are like the main branches of a tree. These bundles of fascicles stay together inside the nerve for a long time before they finally split off and become the major nerves of the body.

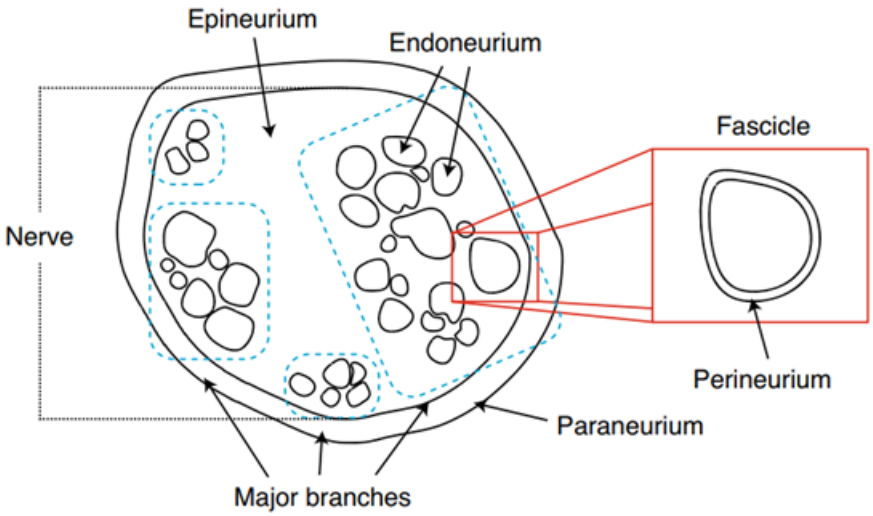


Figure 1.4. Structure of a nerve.

1.5 Neural membrane and Intracellular stimulations

Nerve cells create a range of electrical signals to communicate and store information. Even though neurons aren't naturally great at conducting electricity, they have complex systems in place to produce electrical signals based on how charged particles (ions) move across their outer membranes.
The neural membrane, also known as the cell membrane or plasma membrane, is a vital component of neurons and plays a crucial role in responding to intracellular stimulation and transmitting electrical signals within the cell. It is a semi-permeable lipid bilayer composed mainly of phospholipids. It separates the intracellular environment (inside the neuron) from the extracellular environment (outside the neuron).
Ordinarily, neurons generate a negative potential, called the resting membrane potential. This resting potential is usually around -70 millivolts (mV).
A highly effective method for studying the electrical signals generated by nerve cells is by employing an intracellular microelectrode to gauge the electrical potential across the membrane of neurons [24]. This technique involves inserting a very thin glass tube (microelectrode) into the cell membrane of a neuron. The microelectrode is filled with a good electrical conductor, such as a concentrated salt solution.
One type of these electrical signals is the action potential.It transiently abolishes the usual negative resting potential and causes the electrical charge inside the cell membrane to become positive [24]. Then Action potential travels down the axons of neurons and carries information from one neuron to another [24].
If the intracellular stimulation is strong enough and reaches the threshold, it triggers an action potential. During an action potential, the neural membrane's permeability to ions changes, particularly with the opening of voltage-gated sodium channels. This allows

sodium ions to flow into the neuron, causing depolarization.
Once an action potential is generated, it propagates along the neuron's axon. The neural membrane's properties, including ion channels, ensure the efficient transmission of the action potential.
After an action potential, the neural membrane repolarizes to its resting potential and may even briefly hyperpolarize (become more negative). Ion channels play a crucial role in this repolarization process, returning the membrane to its resting state.
In normal operating conditions, there is no charge transfer between the electrode and the extracellular medium. When there is charge transfer, it is through redox reactions and it leads to corrosion of the electrode.

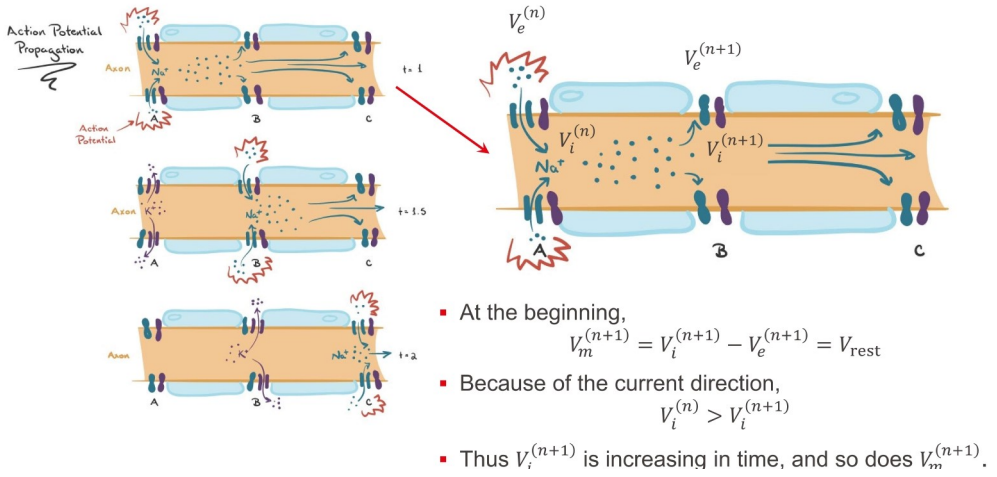


Figure 1.5. Action potential propagation.

1.6 Spinal Cord

The spinal cord is a part of the vertebrate central nervous system. It controls the voluntary muscles of the trunk and upper and lower extremities, serving as a communication pathway between the brain and the rest of the body.
It is a long, thin, tubular structure that extends from the base of the brain, from the brainstem, down the back, and it is situated in the spinal canal, immersed in the cerebrospinal fluid (CSF).
It is enclosed by the vertebral column, but it does not extend the entire length of the vertebral cana [11].
The vertebral column is divided into cervical, thoracic, lumbar, sacral, and coccygeal regions. Also the spinal cord is divided in regions, it has 31 segments: 8 cervical, 12 thoracic or dorsal, 5 lumbar, 5 sacral, and 1 coccygea[11]l.
But, spinal cord and vertebral column levels do not correspond because of their differences in rates of growth during embryonic development , for example the fifth cervical vertebral

body corresponds to the level of the sixth spinal cord segment[11].
The spinal cord is approximately 45 cm long in men and 43 cm long in women[11]. Its width varies, being about 1.27 cm wide in the cervical and lumbar regions, and it widens to about 6.4cm in the thoracic region[11].
Except the first cervical segment, which has only a ventral root, each of the spinal segment has a pair of dorsal and ventral roots and a pair of peripheral nerves called spinal nerves that leave the vertebral canal through the intervertebral foramina[11]. The anterior and posterior roots are formed from groups of nerve fibers called rootlets. The rootlets are attached to the spinal cord by a thin layer of tissue called the pia mater. They are responsible for transmitting signals. Specifically, sensory information, carried by the afferent axons, travels into the spinal cord through the dorsal roots, while motor commands, while motor commands, carried by the efferent axons, exit the spinal cord through the ventral roots. After these roots come together, both sensory and motor axons travel together within the spinal nerves[24] .
The interior of the cord is formed by gray matter, which is surrounded by white matter. Gray matter is concentrated in the central region of the spinal cord, forming an "H" or "butterfly" shape when viewed in cross-section, here the gray matter is conventionally divided into dorsal (posterior) and ventral (anterior) “horns.”
The ventral horns contain the cell bodies of motor neurons, arranged into longitudinal columns, while the dorsal horns contain sensory neurons, organized into layers.
In the thoracic region, the posterolateral portion of the anterior column is called the lateral column. This lateral column contains motor neurons for the autonomic nervous system in the thoracic and upper lumbar areas.
The spinal cord’s white matter is organized into different sections, called columns, each with its own functions.
The dorsal columns handle sensory information, carrying them from the skin, muscles, and joints up to the brain.
The lateral columns contain axon tracts that carry motor signals from the brain down to the muscles. This includes the lateral corticospinal tract, which is the main pathway for voluntary movement.
The ventral columns contain axon tracts that carry sensory information from the internal organs up to the brain, as well as motor signals from the spinal cord down to the muscles.

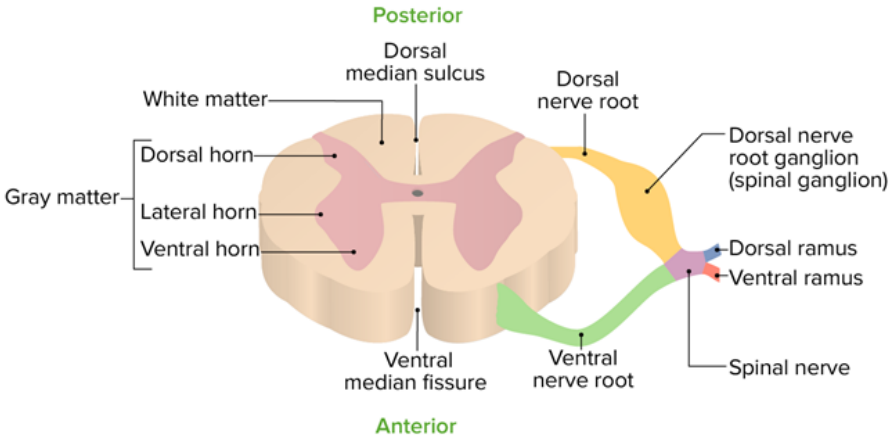


Figure 1.6. Structure of Spinal Cord.

1.7 Vertebrae

Vertebrae are the individual bones that make up the vertebral column. They have a distinct morphology, or structural shape, that is designed to support the body, protect the spinal cord, and facilitate various movements.
The cervical, thoracic, and lumbar vertebrae are similar in structure except for the first (atlas) and second (axis) cervical vertebrae [20].
Each “standard” vertebra is composed of a vertebral body (anterior) and a vertebral arch (posterior) [14].
The atlas is composed of a ring of bone without a body, whereas the axis has an odontoid process around which the atlas rotates [20].
The vertebral body is the anterior part of the vertebra and is typically the largest and weight-bearing component. It is shaped like a cylindrical or somewhat rectangular structure.
The bodies of adjacent vertebrae are separated by a cartilaginous and articulating structure called intervertebral discs which provide the primary support for the column of vertebral bones and permit the required mobility of the spine [20]. The vertebral arch is a bony ring-like structure that extends posteriorly from the vertebral body. It forms the protective vertebral foramen, which houses the spinal cord as it passes through the vertebral column. The arch further divides into two lateral pedicles connected to two posterior laminae, flat plate of bone that connects the spinous process to the vertebral body, a single spinous process [14].
The spinous process is a bony projection that extends dorsally from the vertebral arch. It is the part of the vertebra you can feel when you touch your spine, and two transverse spinous processes that extend laterally at the point where the pedicles are connected to the laminae [14].
"Between each pair of vertebrae there are two openings, the foramina, through which pass a spinal nerve, radicular blood vessels, and the meningeal nerves" [20].

To connect one vertebra to the next there are facet joints that provide stability to the spine.
The morphology of vertebrae can vary between the cervical, thoracic, and lumbar regions. For example, cervical vertebrae often have bifid spinous processes, while thoracic vertebrae have long, downward-pointing spinous processes, and lumbar vertebrae typically have larger bodies for weight-bearing.

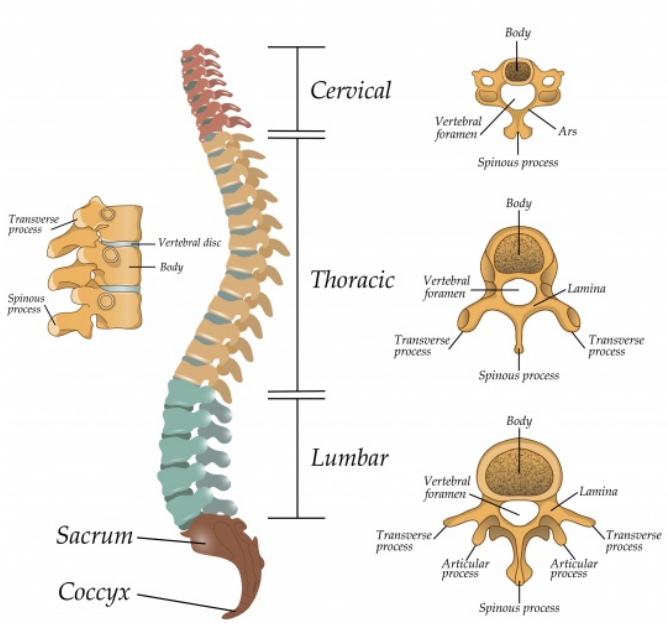


Figure 1.7. Structure of the Spine.

1.8 Spinal Cord Stimulation

Spinal cord stimulation works by implanting a specialized medical device, known as a spinal cord stimulator or neurostimulator into your spine to deliver bursts of electricity in order to alleviate chronic pain. Pain signals travel from your body to your brain through your spinal cord.
The SCS system consists of the generator, leads, and electrodes. The generator is a small battery-powered device implanted under the skin, often in the upper buttock or abdomen. The leads, which are insulated wires, connect the generator to the electrodes.

The electrodes are implanted into the epidural space, which is the area between vertebrae and the outermost membrane of the spinal cord.
Although, the most common stimulation technique id the epidural stimulation, there is also another kind of stimulation of the spinal cord called Transcutaneous stimulation which refers to a medical procedure in which electrical stimulation is applied through the skin to modulate the activity of the spinal cord. It is a non-invasive approach that can

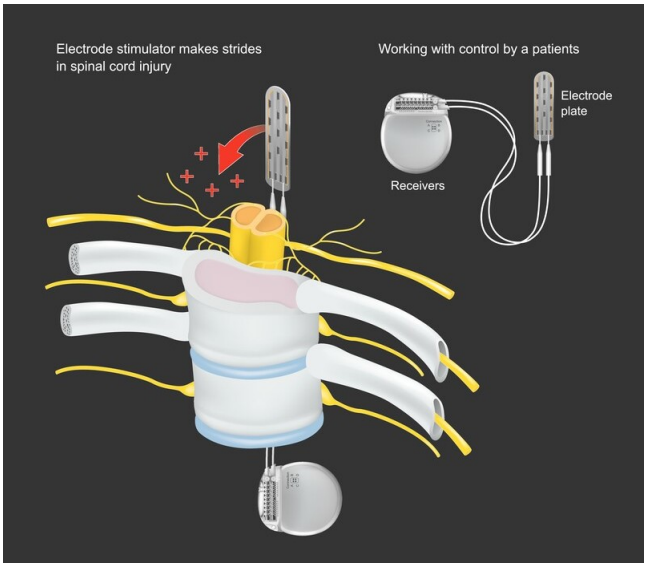


Figure 1.8. Structure of a neurostimulator.

be performed without the need for surgical intervention.
The knowledge gained from computational models of spinal cord stimulation for pain control has led to a rapid expansion of spinal cord stimulation models for other purposes [12]. In experiments SCS computational models have shown that myelinated sensory fibers, responsible for the activation of motor neurons and other cells, are the main targets of spinal cord [12]. Additionally, these models have revealed that using a SCS , the action potentials, generated in primary sensory fibers, disrupt natural sensory feedback, which is crucial for limb movements [9].
Currently, there are still several open questions and challenge regarding these models, such us which anatomical structures, biological and biophysical factor plays a pivotal role on the accuracy of the model [19]. Moreover one open challenge is achieve the automation of manually process using Artificial intelligence-based algorithms which represent breakthrough for the development of in silico models of SCS [19].

1.9 Hybrid Models

Modeling involves the use of mathematical constructs designed to mimic the behavior of a real-world system or a part of it. These mathematical representations are then analyzed and manipulated as if they were the actual physical system. This approach allows us to collect data, conduct experiments, and test hypotheses as if we were working directly with the real system itself. In essence, modeling provides a means to simulate and interact with complex physical systems through mathematical abstractions. Modeling can help us reduce the cost and ethical concerns of experiments, such as animal experimentation, by helping us to Formulate more specific and focused research questions. This helps us to

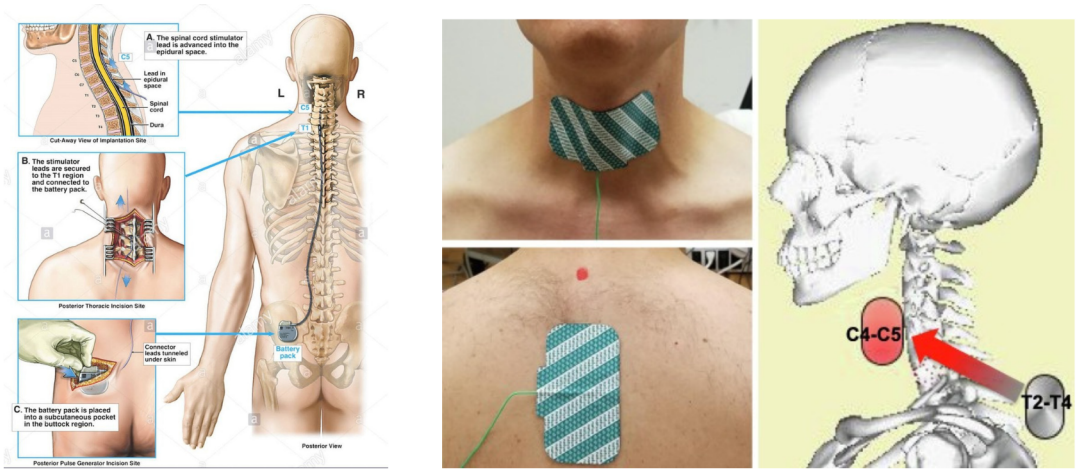


Figure 1.9. On the left: Epidural Stimulation. On the right: Transcutaneous Stimulation

design more efficient and informative experiments. modeling can be thought of as a "denoiser" of reality. It provides a valuable means of making sense of complex experimental results. It enables us to isolate the specific factors we're interested in and work within a controlled, noise-free environment. Moreover, since we have perfect control on model environment, modeling enables us to explore and experiment with different representations of reality, providing valuable information about aspects of the physical world that would otherwise remain beyond our reach for measurement, and allowing us to a to answer questions in fundamental research.
There are various types of models used across different fields of study, and these models serve different purposes and functions. In the field of neuromodulation, modeling the effects of neural stimulation is a fundamental step in the design of neuroprosthetic devices [26] and at the State-of-the-art modelling methods for neuromodulation are based on hybrid models (HM).
Hybrid modeling helps us gauge the resilience and sensitivity of experiments in complex systems in fact we can evaluate how well an experimental setup performs when some partially controllable parameters, such as the electrode surgical insertion coordinates, change. Additionally, by comparing experimental outcomes under various configurations of the given phenomenon, we can also deduce the acceptable ranges for noncontrollable parameters as nerve morphology and topography.
HMs aim to solve two problems, independent one from the other: a volume conduction problem and a neural response determination. The volume conduction is the determination of the electric potential induced by a stimulating electrode in a biological structure for example compute how the electrode current injection modifies the fibers' extracellular medium [26]. This problem arises because electrical signals generated by biological sources, such as neurons or muscles, can spread throughout the surrounding tissue or body, affecting the signals detected by electrodes or sensors. To solve this problem the

finite element modeling (FEM) is normally employed.
FEM is a numerical technique used for solving partial differential equations, the Poisson equation, over complex physical systems with given boundary conditions [26]. FEM begins with the mesh phase, which consists of dividing a complex geometry into a finite number of smaller elements with a simple geometric shape, like triangles, that approximates the complex shape of the domain. After meshing, the mathematical problem is discretized by approximating the solution within each element using a set of basic functions. These basis functions are often piecewise defined, allowing them to represent different functions within each element. The local element equations are assembled into a global system of equations. The system of equations is solved, yielding the values of the unknown variables at discrete points within the domain.
The neural response determination is the prediction of the electric potential induced consequences for single-neuron responses, for example how fibers respond to the imposed extracellular medium [26]. We are interested in tracking the changes in membrane potential over time at a significant distance from the site of neural stimulation. To do this, we typically use simplified electrical models that represent the neuron's behavior. These models divide the length of fibers into smaller segments and simplify them into cable-like structures with multiple compartments.

1.10 UNet

At the state of the art, HMs require a high computational cost and many model evaluations in order to optimize parameters. To address these limitations to some extent, we can turn to surrogate models.
A surrogate model is a simplified system that accurately mimics the behavior of the original model consequently, it can achieve the same results with significantly less computational resources and time. We use a surrogate model based on a 3D UNet. 3D UNet is a three-dimensional extension of the U-Net architecture, which is a convolutional neural network (CNN) commonly used for image segmentation tasks, particularly in medical image analysis.
The architecture of U-Net is symmetric and consists of two major parts: the feature extraction part and the upsampling part [27]. The first part, also known as contracting path, is responsible for identifying the relevant features in the input image. Here, the encoder layers, applying convolutional, rectified linear unit (ReLU) and max pooling operations, take the input data [27] and gradually transform it into more abstract and detailed representations. This process is similar to how feedforward layers work in other convolutional neural networks. On the other hand, the upsampling part, also called the expansive path, works on the abstract representations learned by the contracting path and producing a segmentation map or an output that has the same spatial dimensions as the original input.

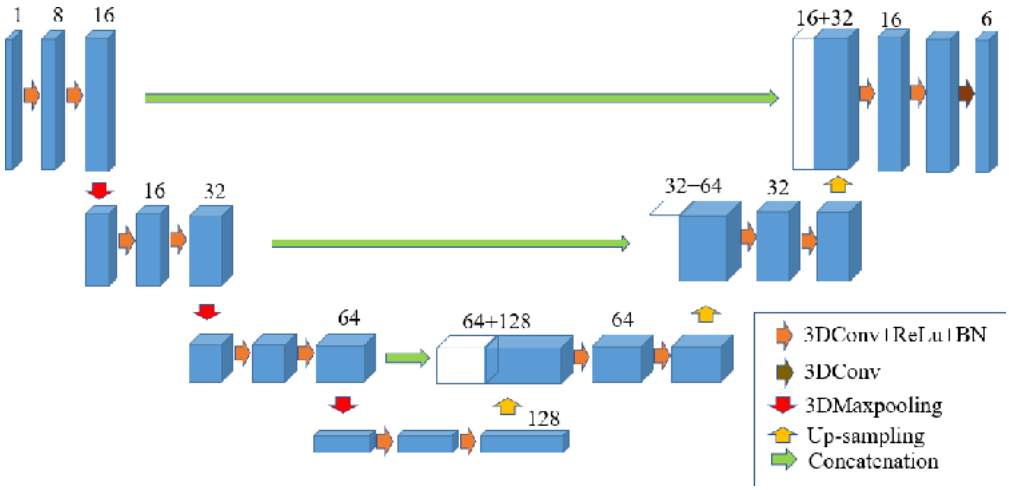


Figure 1.10. Structure of the UNet.

Part II

Chapter2

Chapter 2

Materials and Methods

The purpose of this Chapter is to describe the materials and methods implemented throughout this thesis. For the sake of clarity, the chapter will be divided into four sections. The first will be dedicated to the nerves with straight fascicles model, the second to nerves with complex morphologies. Following in the third section, the spinal cord model will be illustrated, and we will end with the fourth section, where the use of surrogate models will be explained.
For the creation of 3D models, the COMSOL tool was used, while all the code was written in Python.
The entire project has been meticulously documented with examples and explanations using Sphinx [5], an open-source documentation generation tool widely used to create technical documentation and static web pages from text sources. Most of the photos in this thesis are taken from it.

2.1 HM Creation

First of all, a brief introduction on the steps followed to generate all the models and the tools used.

2.1.1 HM Workflow

For solving the FEM during the creation of nerve models and for the spinal cord model, we followed the workflow used and described in [26] . It is divided in the following tasks:

1. Simplify the geometry: starting from histology of the entities to be modeled, an effort is made in order to simplify them using geometric primitives that can be meshed [26].

2. Choose the electrode: choose the electrode type, its dimension and determine the electrode-nerve interface geometry [26].

3. Assign Materials: assign the material properties to each geometric component of the model [26].

4. Set boundary conditions: set the physics of the model, such as fixing the electrode currents [26].

5. Generate the Mesh: divide a complex geometry into smaller elements such as triangles in 2D or tetrahedra 3D [26]. Automatic with COMSOL.

6. Solve FEM: compute the electric potential for each mesh node [26]. Automatic with COMSOL.

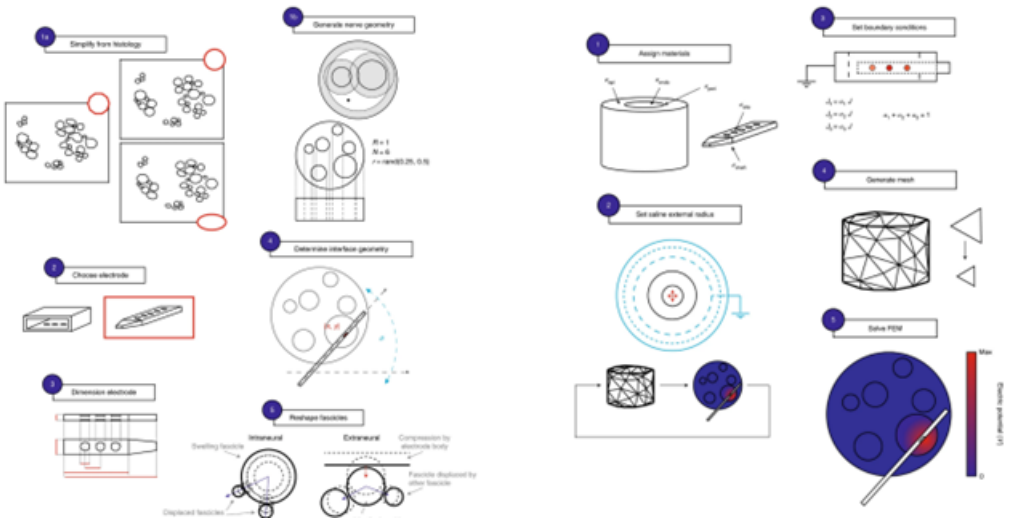


Figure 2.1. HM Workflow.

2.1.2 COMSOL Multiphysics

COMSOL Multiphysics is a multiphysics simulation software tool that allows engineers, scientists, and researchers to model and simulate a wide range of physical phenomena. It provides a versatile environment for solving complex physics-based problems through finite element analysis and it also offers tools for organizing models, and user-friendly features for creating simulation applications.
Additionally COMSOL, through "LiveLink™ for MATLAB, allows to use MATLAB in order to integrate the full capabilities of MATLAB into COMSOL simulations and applications. COMSOL also offers a Java-based interface called COMSOL API, which let you to define all algorithms and data structures of COMSOL model, using Java programming language. [1]

2.2 Nerves with Straight Fascicles

In this initial phase of the thesis, we aim to create a general framework, called HMLab, for the programmatic and automatic generation of nerve models with simple morphologies that is easy to use and easily expandable with additional attributes.
We started from the work presented in the [26], which aimed to create a model capable of solving the finite element problem (FEM) for peripheral nerve stimulation. Here, the construction of the model occurred automatically following the steps presented in paragraph 2.1.1 and making use of COMSOL and 'LiveLink™ for MATLAB', since the chosen programming language was MATLAB.

2.2.1 Geometry

Considering all the main components the nerve is composed of, the proposes the following geometric objects to represent them all in a 3D model made in the COMSOL environment:

- Fascicles: Cylinder OR extrusion of a 2D nerve section approximated as a circle
- Perineurium: 2D contact impedance boundary conditions
- Epineurium: Cylinder OR extrusion of a 2D nerve section approximated as a circle
- Endoneurium: Cylinder

It also adds the saline bath, approximated as a Cylinder, in order to provide a realistic approximation of the intraoperative extraneural space.
Regarding the representations of the electrodes, geometries were chosen that faithfully reproduced their real shape, as their shape is already geometric. For this reason, based on the type of electrode chosen, Cylinders, Blocks or Points were used.

2.2.2 Language and Architecture

From this scenario, we felt the need to reorganize the code to generate a more accessible and usable framework. First of all, we agreed on a change in the chosen programming language, opting for Python instead of MATLAB. To make the code reusable and modular, always following the basic principle of computer science 'Keep It Simple Stupid' (KISS), we agreed that an object-oriented programming language was the best choice. MATLAB, despite its great flexibility, is not originally designed as an OOP language, as it is primarily used for numerical computing. Furthermore, the need to reduce the computational cost of the HM models using machine learning-based surrogate models further encouraged us to look for a language commonly used in machine learning techniques.
Therefore, the choice fell on Python, an OOP language widely used in the machine learning and biomedical engineering fields.

First of all, we translated and restructured the code into Python.
The architecture of HMLab, consists of several classes, with the main one being *FEMModel()*, which contains all the methods to generate the objects that make up the model

and to operate the model itself.
The primary objects in the model for nerve stimulation with straight fascicles are: the nerve, the fascicles, the electrode, and the saline solution. For each of these entities, a class has been defined. Specifically, we have the *Nerve()* class, the *Topography()* class for generating fascicles, the *Electrode* class, and the *Saline()* class.
With the exception of the *Saline()* class, the others are abstract classes aimed at defining a common interface for the classes derived from them.
Specifically, HMLab allows you to choose different types of electrodes, including *Time()* and *Cuff()* types. It also offers the option to choose between an empty nerve, *EmptyNerve()*, and one with fascicles inside, *PeripheralNerve()*. The latter comes with various available fascicles geometries, including polylinear or circular morphology (see the Appendix B for all the available options).
Figure 2.2. provides a more detailed overview of the overall architecture of the entire framework. While the architecture of the specific classes for the nerve model discussed, is presented in Figure 2.3..

HMLab Architecture

HMLab is divided in several classes and subclasses:

- FEMModel : is the main class and contains all the objects needed in the model

1.We can choose between:

- 1a Motor Nerve: It is the class responsible for generating the motor nerve with the chosen morphology.You can find a more detailed description for all the Motor Nerve classes here : Motor Nerve
- 1b Spinal Cord: It is the class responsible for generating the Spinal Cord.You can find a more detailed description for all the Spinal Cord classes here : SpinalCord_Classes

2.We can choose different kind and number of electrodes:

- Implant: It is the class responsible for generating the chosen number of electrodes
- Electrode: It is the class responsible for generating the chosen electrode
- **2a The electrode for a motor nerve stimulation to choose from are:**
  - SoftCuff:It is the class responsible for generating a SoftCuff electrode
  - TIME:It is the class responsible for generating a TIME electrode
  - PointSources:It is the class responsible for generating a PointSources electrode
  - GroundedTIME:It is the class responsible for generating a GroundedTIME electrode
  - CortecCuffCyl:It is the class responsible for generating a CortecCuffCyl electrode
  - CortecCuff:It is the class responsible for generating a CortecCuff electrode
- **2b The electrode for the spinal cord stimulation to choose from are:**
  - TranscutaneousElectrode:It is the class responsible for generating a Block electrode
  - PointSources:It is the class responsible for generating a PointSources electrode

3.We can generate the external environment:

- **3a For the motor nerve:**
  - Saline: It is the class responsible for generating the saline bath
  - InfiniteSaline: It is the class responsible for generating the saline bath with fixed ground
- **3b For the Spinal Cord:**
  - Body: It is the class responsible for generating the biological layers surrounding the spinal cord.

- Motor Nerve Architecture
  - All Nerve Classes
- Spinal Cord Architecture

Figure 2.2. HMLab Architecture.

/ HMLab Overview / Motor Nerve Architecture

Motor Nerve Architecture

- Motor Nerve: It is the class responsible for generating the motor nerve with the chosen morphology

  The morphologies to choose from are:

  - EmptyNerve: It is the class responsible for generating a basic nerve without fascilces
  - PeripheralNerve: It is the class responsible for generating a peripheral nerve

    For the PeripheralNerve we can choose the topography of the fascicles inside the nerve:

    - Topography:It is the class responsible for generating a fascicle topography

    The topographies to choose from are:

    - PolylinearFascicleTopography:It is the class responsible for generating a nerve with polylinear fascicles
    - CircularFascicleTopography:It is the class responsible for generating a nerve with circular fascicles
    - RotatingFascicles:It is the class responsible for generating a nerve with rotating fascicles
    - MergeFasc:It is the class responsible for generating a nerve with splitting and merging fascicles

Figure 2.3. HMLab Architecture of Nerve Classes.

Every class, except the main one, is structured according to a skeleton that is organized in this way:

- Class-specific attributes
- Methods common to all objects, defined according to the entity's peculiarities
- Class-specific methods

See Figure 2.4. for a view of classes skeleton.

/ HMLab Classes

HMLab Classes

Classes Skeleton

Every class, except the main one, is structured according to a skeleton that is organized in this way:

def __init__(self):

Specify the characteristic attributes of the object

Parameters: **self** – the caller object

def get_params(self):

Assign object properties employing the GUI

Parameters: **self** – the caller object
Returns: the modified caller object

def add_to_model(self, model,...):

Add the object to the model

Parameters:
- **self** – the caller object
- **model** – the COMSOL model to modify
- **others** – other parameters

Returns: the modified COMSOL model

def assign_materials(self, model):

Assign a material to the object

Parameters:
- **self** – the caller object
- **model** – the COMSOL model to modify

Returns: the modified COMSOL model

def other_methods(....):

Methods specific to the class

Figure 2.4. HMLab Class Skeleton.

2.2.3 Getting Started

First of all, we have to set the COMSOL model nodes for Hybrid Modelling, define a 3D geometry, the electric conduction physics, a stationary study, and an automatically generated mesh. To perform these tasks, we rely on the use of COMSOL classes (Figure 2.5.)

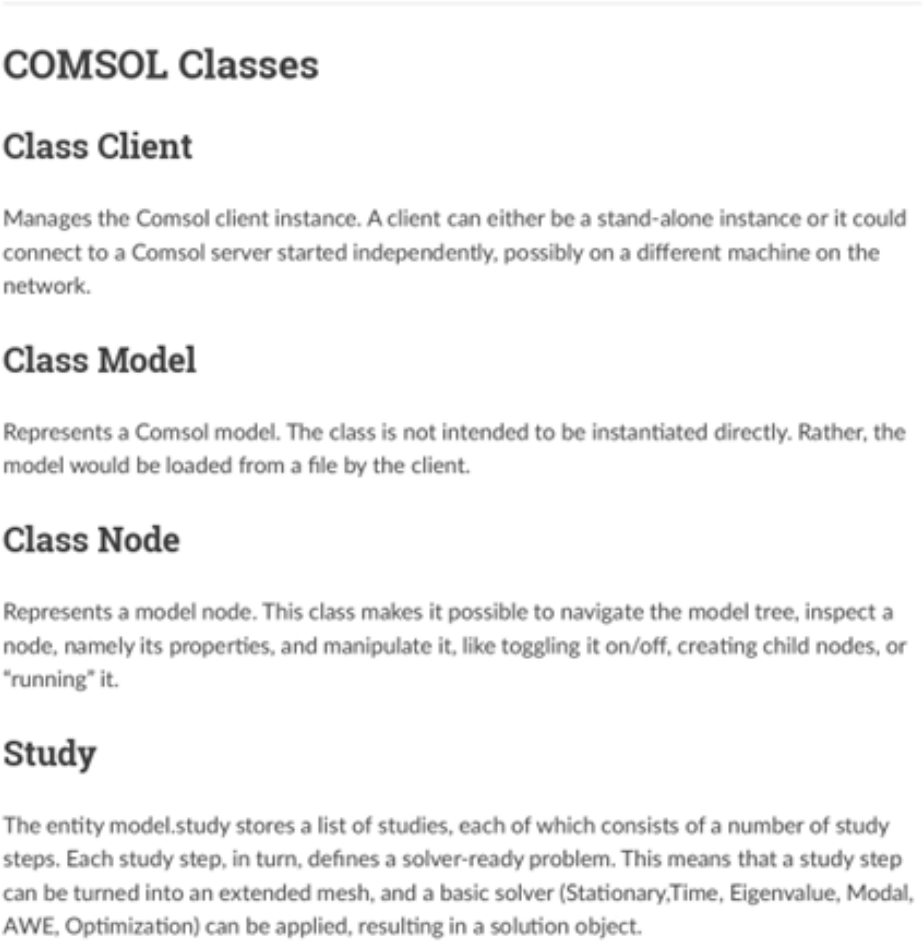


Figure 2.5. COMSOL Classes.

To create these classes and execute the previously mentioned tasks, the methods, illustrated in Figure 2.6., are executed in the main class.
Specifically, some of these methods are provided by MPh, a Python-based scripting interface for COMSOL, which makes easier to use COMSOL with Python [3].

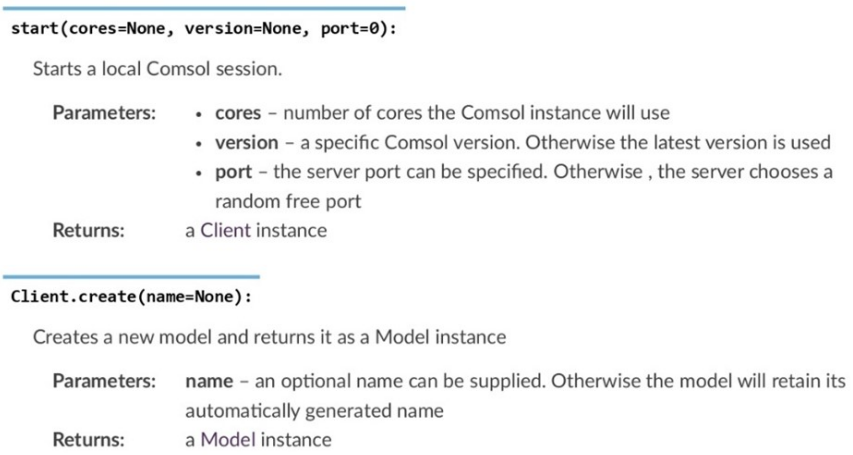


Figure 2.6. COMSOL Methods.

Unfortunately, the methods provided by MPh are not sufficient to fulfill all the steps, so we resort to using the Java COMSOL API [8]. To access the Java COMSOL API, we need to call model.java, which corresponds to the Java object encapsulated within this instance. Figure 2.7. shows these Java methods
All the previously shown methods are invoked through the main class *FEMModel()*.

`model.java.modelNode().create(<tag>):`

Creates a Node with the given tag

Parameters: **tag** – a tag can be supplied
Returns: a Node instance

`model.java.component(<Nodetag>).geom().create(<tag>,<sdim>):`

Creates creates a geometry in the specified Node

Parameters:
- **Nodetag** – tag of the Node
- **tag** – a tag to assign
- **sdim** – a space dimension of the geometry

Returns: a geometry

`model.java.component(<Nodetag>).physics().create(<tag>,physint,<geomtag>):`

Adds a physics interface to the model and initializes it with defaults

Parameters:
- **tag** – a tag to assign
- **physint** – specifies which physics interface to create
- **geomtag** – specifies the geometry in which we add the physics interface

`model.java.study().create(<tag>):`

Creates a new study sequence

Parameters: **tag** – a tag to assign to the study

`model.java.study(<tag>).feature().create(<ftag>,type):`

Creates a new study step of the given type within the specified sequence

Parameters:
- **tag** – the tag of the study
- **ftag** – a tag to assign to the study step
- **type** – the type of the study

`model.java.study(<tag>).feature(<ftag>).activate(<physpath>,<bool>):`

Activates or deactivates a physics interface or a physics feature

Parameters:
- **tag** – the tag of the study
- **ftag** – the tag of the study step
- **physpath** – the path to a node in a physics interface
- **bool** – a boolean value to activate or deactivate the physics interface

`model.java.mesh().create(<tag>,<gtag>):`

Creates a meshing sequence for the geometry

Parameters:
- **tag** – a tag to assign
- **gtag** – the tag of the geometry

`model.java.mesh(<tag>).autoMeshSize(<size>):`

Adjusts the overall size of a physics-induced mesh

Parameters:
- **tag** – the tag of the mesh
- **size** – the size to assign to the mesh

Figure 2.7. Java methods to create the model.

2.2.4 Create the Model

Once the COMSOL environment setup is complete, we proceed to create the actual 3D model. Through *FEMModel()* class, the following methods are called, which allow the creation of the physical model, following the steps previously described.
In particular the methods called are:

- `assign_materials()`: assign materials to the geometrical entities in the COMSOL model
- `characterize_all_sites()`: run a single-site, unit current FEM for each active site in the implant
- `generate_geometry()`: generate the model geometry by adding nerve, implant and saline geometries to the model
- `generate_materials()`: create materials with their respective physical properties
- `get_params()`: assign object properties employing the GUI
- `get_custom_params()`: customize the object with the parameters in input
- `get_nerve_params()`: generate Nerve object and call its `get_params()`
- `get_implant_params()`: generate Implant object and call its `get_params()`
- `get_saline_params()`: generate Saline object and call its `get_params()`

In detail, let's analyze the various steps to complete all the tasks.
Once the geometric primitives to use for representing the biological entities are selected, these are created and added to the model using the function `generate_geometry()`, which inside call `add_to_model()` for each object present in the model.
Figure 2.8. shows an example of how this method works, creating an electrode TIME.

Add to Model

To create these geometry we can use the main method `generate_geometry()`, present in FEMModel class,which inside calls the method `add_to_model()` for all the objects of the model. One of the necessary methods for creating an object in COMSOL is the following:

`model.java.geom(<tag>).feature().create(<ftag>,type):`

Adds a geometry object in the current geometry

Parameters:
- **tag** – geometry tag
- **ftag** – geometry feature tag
- **type** – feature type such as *Block*, *Cylinder*

Here an example of creation of a piece of the geometry for a TIME electrode

```
def add_to_model(self, model):
        """   ADD_TO_MODEL add the electrode to the model

                Args:
                    self : the caller object
                    model : the COMSOL model to modify

                Returns:
                    model : the modified COMSOL model
        """
        print('Generating electrode geometry in electrode reference frame')
        # Generate electrode body in electrode reference frame
        geom1 = model.java.geom('geom1')
        elec_full = geom1.create(str('elec_full_'+ str(self.id)), 'Block')
        elec_full.set('size', [str(self.l_shaft), str(self.h_shaft),str(self.w_shaft)])
        elec_full.set('base', 'center')
        elec_full.set('createselection', 'on')

        geom1.run()
```

Figure 2.8. Creation of electrode TIME.

2.2.5 Assign Materials

In more detail, we can analyze the task related to material assignment, which occurs through several steps.
First, materials are selected and generated with their respective properties, and then the assignment takes place.
To generate the materials, we have implemented the method `generate_materials()`, which consists of calling the following Java COMSOL API :

- `model.java.material().create()`: create a new material for the current geometry [8]

- `model.java.material().propertyGroup(<grouptag>).set(<pname>,<expr>)`: set the expression for the given material property [8]

The materials used and created are listed in Table 2.1.

Tag	Name	Relpermittivity	Electricconductivity
endo	Endoneurium	80	0.083-0-0-0-0.083-0-0-0-0.571
peri	Perineurium	80	0.0009
epi	Epineurium	80	0.083
fib	Fibrosis	80	0.1
sal	Saline	80	2
dura	Dura Mater	1	0.0025
arac	Arachnoid	1	2
pia	Pia Mater	1	0.016
as	Sites	1	1.0e-10
elec	Elecshaft	1	1.0e-10

Table 2.1. Materials for Nerve Model

For the assignment, we have implemented the method `assign_materials()`, which consists the following steps:

1. Select the entities to wich ypu want to assign the material. To do it we can use Java COMSOL API:
   - `model.java.selection()`: select the domain [8]
   - `model.java.selection(<tag>).entities()`: returns the geometric entities of the selection on the given geometry as an integer array [8]
2. Assign materials to the selected entities. To do it we can use Java COMSOL API:
   - `model.java.material(<tagmat>).selection().set()`: set a specified material to the entities selected [8]

2.2.6 Choose Electrodes

Select the electrode type, specify its dimensions, and define the geometry of the electrode-nerve interface.
This task is performed using the class Implant().
All the electrode types with their respective parameters are in the Appendix, here we show only the Soft Cuff type (Figure 2.9.)

2.2.7 Set Boundary Conditions

Set the physics of the model, such as fixing the electrode currents and the 2D contact impedance for modelling the Perineurium.

Soft Cuff

In the figure an electrode of type SOFT CUFF is represented using COMSOL object

Parameters:

- radius of the nerve = 1 mm
- length of cuff along the nerve = 1 mm
- thickness of the cuff = 0.2 mm
- active site diameters = 0.05 mm
- distance between active sites = 0.2
- number of active sites = 8 mm
- active site depth = 0.07 mm
- distance between active site rings = 0.1 mm
- z-position of active site = 0
- angle of insertion around z axis = 0

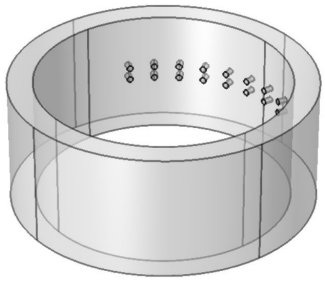


Figure 2.9. Example of Electrode called Soft Cuff.

You can use the following method to assign a current to the electrode`set_active_site_current()`, which consists of calling the following Java COMSOL API :

- `model.java.material().create()`: create a new material for the current geometry [8]
- `model.java.physics(<tag>).feature(<ftag>).set(<pname>,<value>)`: set a parameter value to the feature instance of physics interface [8]

To model the perineurium we use the contact impedance, which is relates to the electrical resistance encountered at the interface between an electrode and the tissue or medium it

is in contact with.
You can follow these steps:

1. Select all the boundaries of each fascicle:. To do it we can use Java COMSOL API:
   - `model.java.component(<tag>).selection().create(<tag>,<type>)`: create a selection of the specified type [8]
   - `model.java.component(<tag>).selection(<tag>).set(property,<value>)`: set a property value for the selection [8]
   - `model.java.component(<tag>).selection(<tag>).entities()`: return the geometric entities of the selection in given geometry as integer array [8]
2. Create and apply contact impedance boundary conditions. To do it we can use Java COMSOL API:
   - `model.java.physics(<tag>).selection().create(ftag>,feature,<dim>`: set new feature instance to the physics interface and initializes the feature [8]

2.2.8 Compute the Mesh and Solve the FEM

At this point, to obtain a functional model, we need to generate the mesh and solve the FEM. These final steps are accomplished using two functions, `solve_single_run()` and `characterize_all_sites()`, present in the *FEMModel()* class.
These functions allow us to connect our scripts to the COMSOL server and delegate these tasks to it.
The methods used by COMSOL:

- `solve(study=None)` : solves the named study, or all of them if none given. [3]
- `evaluate (expression, unit=None, dataset=None, inner=None, outer=None)`: Evaluates an expression and returns the numerical results. We used: evaluate(['V']) where 'V' stands for Potential. [3]
- `model.java.result().export().create(<etag>,<ptag>,etype)`: create and export feature [3]

Once these final steps are completed, we obtain a text file in the .txt format containing the lead field matrix (Figure2.10.), important to comprehend the stimulation. This matrix includes the coordinates of the mesh nodes and their corresponding potential value.

HMLab allows us to compute the potential both at the nodes of the mesh generated by COMSOL and at points with coordinates of our choice. This is done by passing the filename containing coordinates (x, y, z) of nodes as a parameter to the `characterize_all_sites()` function.
In addition to the lead field matrix, we also obtain an .mph file (COMSOL model file) that can be opened in COMSOL for a visual representation of the generated 3D model. This

allows us to interact with the model through the COMSOL GUI for a visual understanding of the results. We can obtain this file invoking the COMSOL API `save(path=None, format=None)`[3].
You can see the entire workflow to solve the FEM in Figure 2.11..

```
% Model:              merge_3sez_4mm_1.mph
% Version:            COMSOL 5.6.0.280
% Date:               Aug 27 2023, 18:01
% Dimension:          3
% Nodes:              262144
% Expressions:        1
% Description:        Potenziale elettrico
% Length unit:        m
% x                   y                   z                   V (V)
-0.001                -0.001              0.001               1.5665582646312168E-6
-0.001                -0.001              0.0011587301587301588 1.7030504665721288E-6
-0.001                -0.001              0.0013174603174603175 1.8645870668179338E-6
-0.001                -0.001              0.001476190476190476 2.0491586868860396E-6
-0.001                -0.001              0.0016349206349206635 2.254175572634125E-6
-0.001                -0.001              0.0017936507936507935 2.4777388034716723E-6
-0.001                -0.001              0.0019523809523809522 2.7154253990917148E-6
-0.001                -0.001              0.002111111111111111 2.957550147004609E-6
-0.001                -0.001              0.00226984126984127 3.1910723978200282E-6
-0.001                -0.001              0.0024285714285714284 3.4042745555975059E-6
-0.001                -0.001              0.002587301587301587 3.5842722024078l9E-6
-0.001                -0.001              0.002746031746031746 3.7113429489446386E-6
-0.001                -0.001              0.0029047619047619043 3.7730195860617722E-6
-0.001                -0.001              0.0030634920634920633 3.7547335093479477E-6
-0.001                -0.001              0.003222222222222222 3.6595651205330084E-6
-0.001                -0.001              0.0033809523809523808 3.4893600529830566E-6
-0.001                -0.001              0.0035396825396825393 3.2592248636906646E-6
-0.001                -0.001              0.003698412698412698 2.9810335563524444E-6
-0.001                -0.001              0.0038571428571428567 2.6869875681606767E-6
-0.001                -0.001              0.004015873015873015 2.4066051100315186E-6
-0.001                -0.001              0.004174603174603175 2.1653334242989873E-6
```

Figure 2.10. Example of lead field matrix.

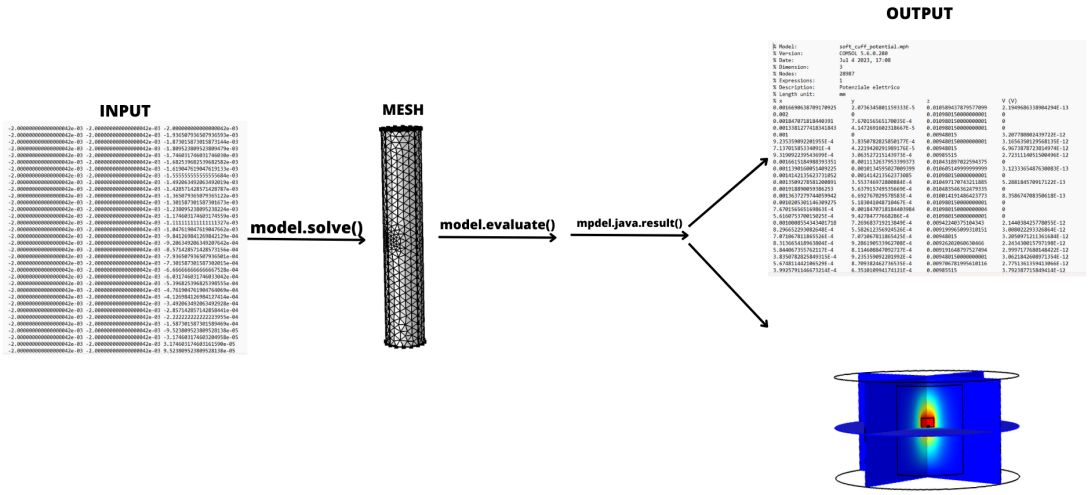


Figure 2.11. Workflow to solve the FEM.

To conclude this section, an example of peripheral nerve stimulation with 4 circular fascicles using two TIME electrodes and a current of 1e-6 A (Figure 2.12.). For more examples see the Appendix C.

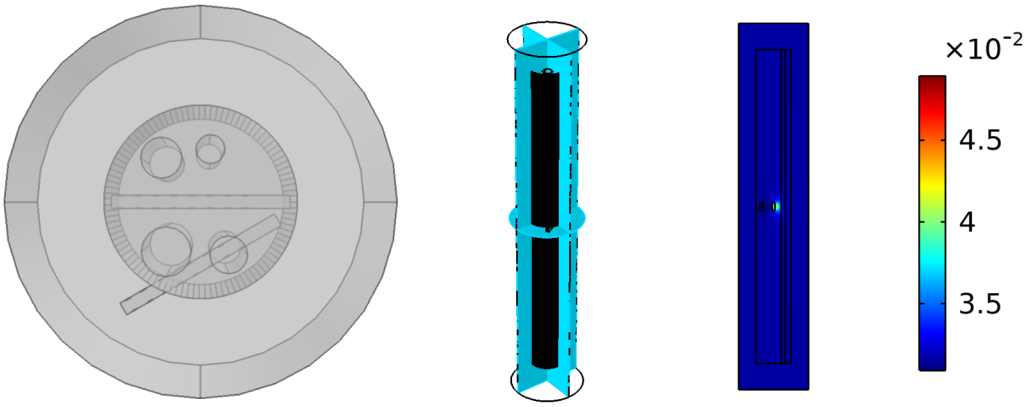


Figure 2.12. Example of peripheral nerve stimulation with 4 circular fascicles using two TIME electrodes and a current of 1e-6 A.

2.3 Nerves with complex morphologies

In this section, we will discuss the creation of 3D models for nerves with rotating fascicles and nerves with fascicles that join and split.
For the creation of these models, we followed the same strategy proposed by HMLab, implemented in the previous section. In fact, we used the same framework and, consequently, the same classes and methods as before, with the addition of specific classes for these new fascicular topographies.
This demonstrates how HMLab is a framework capable of generating even more complex models than the initial one with the simple addition of classes and methods.

Since the biological entities remain the same, the morphology does not vary compared

to the models proposed previously. The only thing that changes is the fascicular topography.
The same tools, COMSOL and Python, have been consistently used.

2.3.1 Fascicles

Fascicles in a nerve are groups of nerve fibers gathered together, surrounded by connective tissue called perineurium, which constitute the basic unit for the transmission of neural signals through the peripheral nervous system. In general, the size of fascicles is an anatomical feature that can vary significantly among different nerves and individuals. For example, thinner nerve structures can have fascicles with diameters often less than a millimeter, while larger nerves may contain fascicles with diameters of several millimeters. As a result, it becomes challenging to have an accurate knowledge of their topography, which depends on the function that the fascicles perform and their location in the nervous system. We know that there are parallel fascicles, intertwined fascicles, branched fascicles, and mixed fascicles.
Consequently, we felt the need to create stimulation models that represented more complex topographies than the straight one presented in the previous section. In particular we modeled rotating fascicles and merging and splitting fascicles.

2.3.2 Geometry

Starting from the geometric objects used in Section 2.2.1 to describe a nerve, we have decided to make the following changes regarding the choice of geometric primitives:

- Perineurium: Loft of a Circle.
- Epineurium: Loft of a Polygon.
- Endoneurium: Loft of a Circle
- Saline Bath: Cylinder
- Electrodes: Blocks or Cylinders or Points

Since COMSOL does not allow for the creation of an entity that rotates with random angles changing along its path, or entities that fork and then converge, we have opted to divide the nerve into multiple sections of the same length.
At this point, we have created the fascicles using the COMSOL object called "Loft" [2]. In fact, in COMSOL, the "Loft" is used to create a smooth, continuous surface or solid shape by connecting multiple cross-section profiles. It is useful to create complex shapes that smoothly transition between different cross-sectional profiles [2].
We have chosen a circular morphology for our fascicles, and consequently, we have used the same data (coordinates for the epineurium, center, and radius of the endoneurium) used for generating the nerves with straight circular fascicles. This implies using circles for the endoneurium and polygons for epineurium as profiles from which the lofted structures were generated.

Another difference from the straight fascicles is the implementation of the perineurium. Indeed, since COMSOL does not allow the application of contact impedance to curved structures, we have decided to implement the perineurium in the same way as the endoneurium but with a diameter 6% larger (Figure 2.13.).

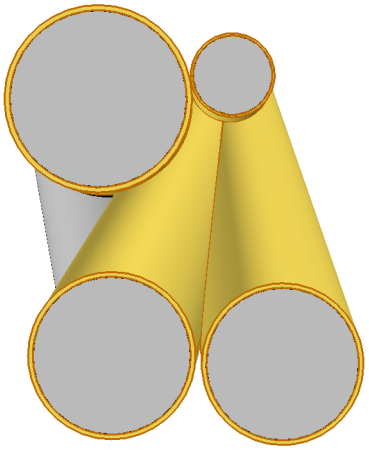


Figure 2.13. The yellow loft represent the perinerium while the grey one the endonerium.

2.3.3 Rotating Fascicles

For the rotating fascicles,we create a new class called 'RotatingFascicles()' with all the methods described in HMLab. The architecture overview is shown in the Figure 2.14.

We use the geometry we analyzed before adding some details. To the circular profiles used for the Lofts generation, a basic rotation of 30 degrees plus a random delta ranging from 0 to 20 degrees was applied, section by section. These latter parameters were experimentally determined to ensure that the four fascicles in the model do not intersect.

Taking a closer look at the implementation of this model, we can observe that whenever objects are created inside other objects, such as the endoneurium wrapped by the perineurium, surrounded by the epineurium, which is then immersed in the saline solution, a series of COMSOL operations must be performed. Specifically, if two objects touch each other, their intersection must be calculated and removed from one of the objects in question to avoid duplicates. The operations that enable us to achieve this result are "Intersection" and "Difference" [2].

All these operations increase the complexity of the model, resulting in an increased computational load and difficulty in mesh generation. Exactly this difficulty with mesh generation has led us to impose a constrain on this model.Specifically, to overcome this inconvenience and obtain a functional model, we decided to simplify the geometry by imposing in advance that the rotating fascicles would rotate at a random angle but within a certain range of values to ensure that they did not intersect. This decision was made both to address the problem described earlier and because these potential intersections

RotatingFascicles module

class **RotatingFascicles.RotatingFascicles**

Bases: `Nerve`

Subclass of Nerve class representing Peripheral Nerve with Rotating Fascicles

add_to_model(*model*, *saline*)

ADD_TO_MODEL add the nerve and the saline to the model

Args:

self : the caller object

model : the COMSOL model to modify

saline : the saline object

Returns:

model : the modified COMSOL model

assign_materials(*model*)

ASSIGN_MATERIALS assign a material to the nerve

Args:

self : the caller object

model : the COMSOL model to modify

Returns:

model : the modified COMSOL model

get_params()

GET_PARAMS assign object properties employing the GUI

Args:

self : the caller object

Returns:

self : the modified caller object

Figure 2.14. Architecture of RotatingFascicles Class.

were not of particular interest for the model we wanted to create.
A model with rotating fascicles is visible in the Figure 2.15., in this example we used 4 sections to compose the nerve.

To complete the model and make it usable for simulating stimulation, we needed to add electrodes. In the example shown in the Figure 2.19., PointSources electrodes were created, specifically four grids of 16 electrodes each. This step was carried out by calling

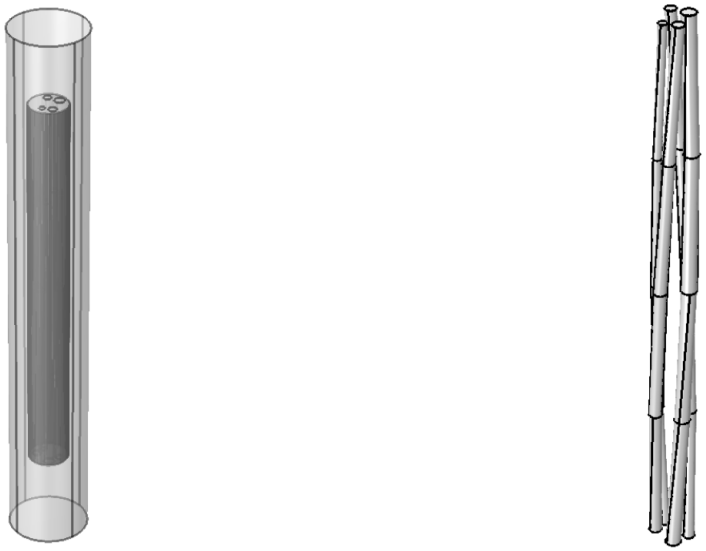


Figure 2.15. A nerve model with rotating fascicles.

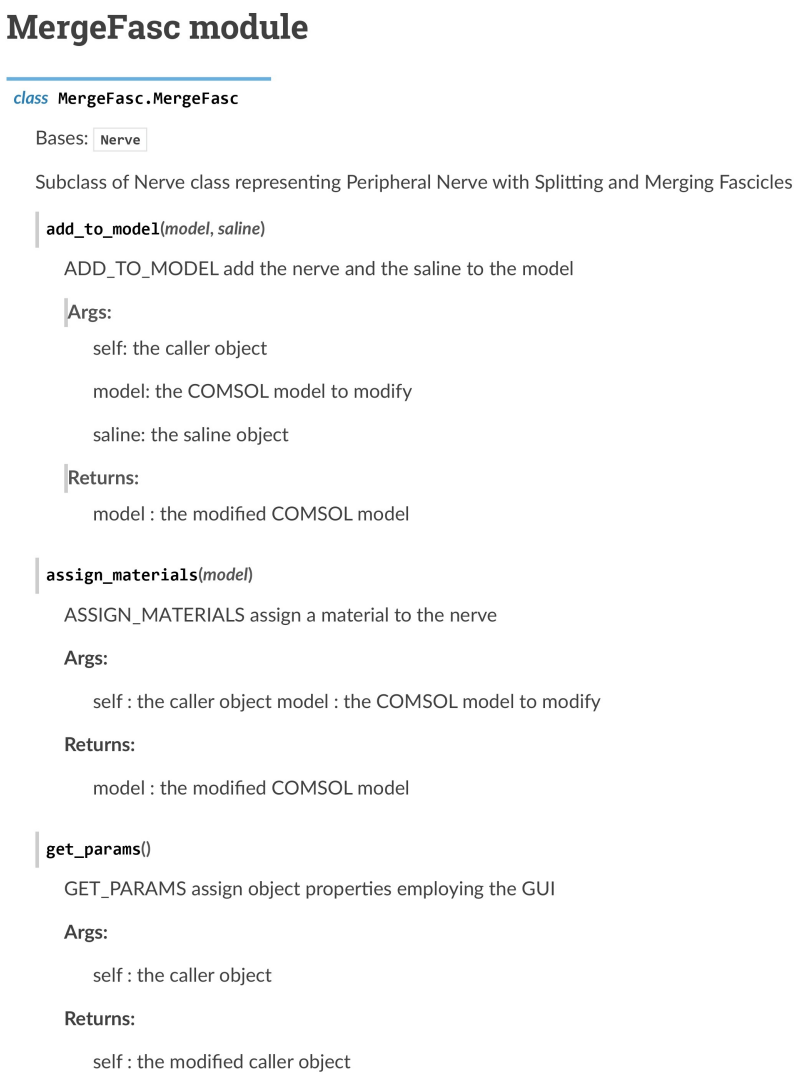
the *Implant()* class first, through the main *FemModel()* class, to decide how many and which electrodes to create, and then the *PointSources()* class.
To create grids at predefined heights, we added the `add_to_mode_height()` method in Implant(), which takes as input the vector with the heights at which the grid should be created.
In PointSources(), we also added a method `add_to_model_height()` that modifies the original `add_to_model()`, as it requires the input parameter representing the height at which to generate the grid of points.

2.3.4 Splitting and Merging Fascicles

For this model, we have created a class called *MergeFasc()* that implements all the basic methods described in the presentation of our framework. The architecture overview is

shown in the Figure 2.16..

We have used the same geometric approximations illustrated in section 2.3.2. In this



MergeFasc module

class `MergeFasc.MergeFasc`

Bases: `Nerve`

Subclass of Nerve class representing Peripheral Nerve with Splitting and Merging Fascicles

`add_to_model`(*model*, *saline*)

ADD_TO_MODEL add the nerve and the saline to the model

Args:

self: the caller object

model: the COMSOL model to modify

saline: the saline object

Returns:

model : the modified COMSOL model

`assign_materials`(*model*)

ASSIGN_MATERIALS assign a material to the nerve

Args:

self : the caller object model : the COMSOL model to modify

Returns:

model : the modified COMSOL model

`get_params`()

GET_PARAMS assign object properties employing the GUI

Args:

self : the caller object

Returns:

self : the modified caller object

Figure 2.16. Architecture of MergeFasc Class.

case, our fascicles can split and subsequently join different fascicles. For this model, a new parameter has been added, a list containing a number of sublists equal to the number of sections, inside which there are the fascicles numbers indicating the endpoint of the fascicles sublists index, as shown more clearly in the Figure 2.17..

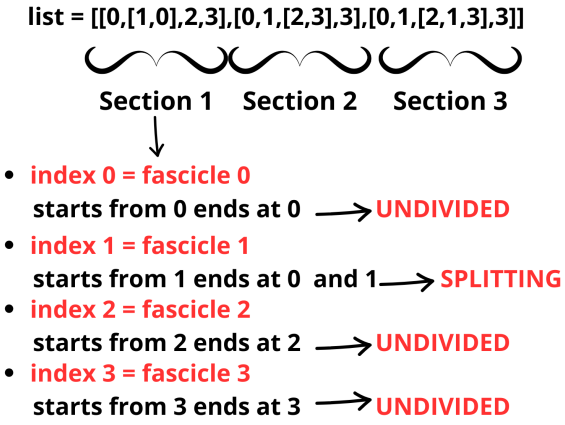


Figure 2.17. List describing the fascicles paths.

The function created here to generate the model is more complex than the others because, first, for each section, it needs to separate the construction of fascicles that either merge or diverge from those that remain straight. We perform this check by looking at the length of the list corresponding to the fascicle under analysis. If length is 1, it means the fascicle undergoes no changes. In this model, fascicles that completely merge into another fascicle have not been considered.
Then, we have divided the function into two parts. One is dedicated to the fascicles that remain intact, and the other for fascicles that undergo mutations. For the fascicles that do not divide, endoneurium and perineurium have been created using lofts, as shown for the other models.
We provide a detailed view of all the steps:

1. Create inferior loft section:

```
wpn = geom1.feature().create(str('wpn_'+str(k)+'_sez_base'+str(j)), 'WorkPlane')
wpn.set('planetype', 'quick')
wpn.set('quickplane', 'xy')
wpn.set('quickz', (str(ids)))
```

2. Create Endoneurium base section:

```
x = self.fascicles[k, 0]
y = self.fascicles[k, 1]
r = radius_fasc[k]
endo = wpn.geom().feature().create(str('fascicles_base_'+ str(k)+'_sez_'+ str(j)), 'Circle')
endo.set('x', x)
endo.set('y', y)
endo.set('r', r)
wpns.append(str('wpn_'+str(k)+'_sez_base'+str(j)+'.'+'fascicles_base_'+ str(k)+'_sez_'+ str(j)))
```

3. Create Perineurium base section:

```
imp = wpn.geom().feature().create(str('perinevrio_'+ str(k)+'_sez_'+ str(j)), 'Circle')
imp.set('x', x)
imp.set('y', y)
imp.set('r', r+((r*6)/100))
imps.append(str('wpn_'+str(k)+'_sez_base'+str(j)+'.'+str('perinevrio_'+ str(k)+'_sez_'+
str(j))))
```

4. Create superior loft section:

```
wpn = geom1.feature().create(str('wpn_'+str(k)+'_sez_'+str(j)), 'WorkPlane')
wpn.set('planetype', 'quick')
wpn.set('quickplane', 'xy')
wpn.set('quickz', str(self.zet[j]))
```

5. Create Endoneurium superior section:

```
x_up = self.fascicles[self.sez[j][k]][0]
y_up = self.fascicles[self.sez[j][k]][1]
r_up = compute_radius(self.sez[j][k], union_fasc, radius_fasc)
radius_fasc_up[self.sez[j][k]] = r_up
endo = wpn.geom().feature().create(str('fascicles_'+ str(k) +'_sez_'+ str(j+1)), 'Circle')
endo.set('x', x_up)
endo.set('y', y_up)
endo.set('r', r_up)
```

6. Create Perineurium superior section:

```
imp = wpn.geom().feature().create(str('perinevrio_'+ str(k) +'_sez_'+ str(j+1)), 'Circle')
imp.set('x', x_up)
imp.set('y', y_up)
imp.set('r', r_up+((r_up*6)/100))
```

7. Create all lofts:

```
imps.append(str('wpn_'+str(k)+'_sez_'+str(j)+'.'+ 'perinevrio_'+ str(k) +'_sez_'+ str(j+1)))
wpns.append(str('wpn_'+str(k)+'_sez_'+str(j)+'.'+ 'fascicles_'+ str(k) +'_sez_'+ str(j+1)))

##create loft fascicle##
loft = geom1.feature().create("loft_sez"+str(j)+'_fasc_'+str(self.sez[j][k])+'_'+str(k) ,
"Loft")
loft.selection("profile").set(wpns)

##Loft Perineurium ##
loft = geom1.feature().create("loft_per_sez_"+str(j)+'_fasc_'+str(self.sez[j][k])+'_'+str(k) ,
"Loft")
loft.selection("profile").set(imps)
loft.set('createselection','on')
```

In the second part, the more complex one, as many lofts as the arriving fascicles were created, starting from the fascicle that initially divided. For example, if fascicle number 1 divides into 3 and then merges with fascicles 2, 3, and 4, three lofts will be created. For each loft, the base is always the circle representing the starting fascicle, '1,' and the endpoint circles are those of the three arriving fascicles, '2,' '3,' and '4'.
Their intersection is calculated using the COMSOL method [2]:
`model.component(<ctag>).geom(<tag>).create(<ftag>,"Intersection")`, which is then subtracted from all entities except one, using the COMSOL method [2] :
`model.component(<ctag>).geom(<tag>).create(<ftag>,"Difference")` .
Then, to create a single entity for each fascicle, we merged these lofts using the COMSOL method [2]:
`model.component(<ctag>).geom(<tag>).create(<ftag>,"Union")`.

Since the construction of the lofts remains the same, only the number of fascicles created changes, we focus on the various intersections, differences, and unions.

- Intersection:

```
inputs.append(split_loft_imp[0])
inputs.append(split_loft_imp[1])
loft_int_fasc = geom1.create('int_peri_' +str(k)+'_sez_'+str(j)+'_num_0', 'Intersection')
loft_int_fasc.selection('input').set(inputs)
loft_int_fasc.set('createselection','on')
```

- Difference:

```
loft_diff =geom1.create('diff_peri_' +str(k)+'_sez_'+str(j)+'_num_0', 'Difference')
loft_diff.selection('input').set((peri_list[0]))
loft_diff.selection('input2').set('int_peri_' +str(k)+'_sez_'+str(j)+'_num_0')
loft_diff.set('createselection','on')
```

- Union:

```
endogeom = geom1.feature().create('endogeom_peri_'+str(k)+'_sez_'+str(j)+'_num_1', 'Union')
endogeom.selection('input').set(input_union)
endogeom.set('createselection','on')
union_peri_split.append('endogeom_peri_'+str(k)+'_sez_'+str(j)+'_num_1')
```

The examples above show the operations performed only once on the perineurium, but the same operations were also carried out on the endoneurium and multiple times.
At this point, once the construction of all the fascicles in a section is completed, intersections need to be calculated, and consequently, differences need to be computed, for all the fascicles that have merged into the same final fascicle. To do this we used the same operations as before: Intersection, Difference and Union.
Furthermore, an additional function has been created for calculating the occurrences of a specific fascicle in each section, a useful function for identifying all intersections in the incoming fascicle during the final phase of section construction. Again, we provide the piece of code:

```
def count_occ_sez(j,sez):
union_fasc={ 0: [], 1:[], 2:[], 3:[]}
for i in range (len(sez[j])):
        for k in range (4):
                if np.size(sez[j][i])==1:
                        if sez[j][i] == k:
                                union_fasc[k].append(i)
                else :
                        for x in sez[j][i]:
                                if x == k:
                                        union_fasc[k].append(i)
 return union_fasc
```

We also create a function useful for calculating the radius of the circular section of the incoming fascicle. It should be noted that whenever two fascicles merge, they will create a larger entity than the one they started with (Figure 2.18.). We also provide the code for this, for further clarity:

```
def compute_radius(id_fasc, union_fasc,radius_fasc):
        Area = 0
        x = union_fasc[id_fasc]
        for j in x:
           Area = (radius_fasc[j]**2)*np.pi + Area
        radius = np.sqrt(Area/np.pi)
        return radius
```

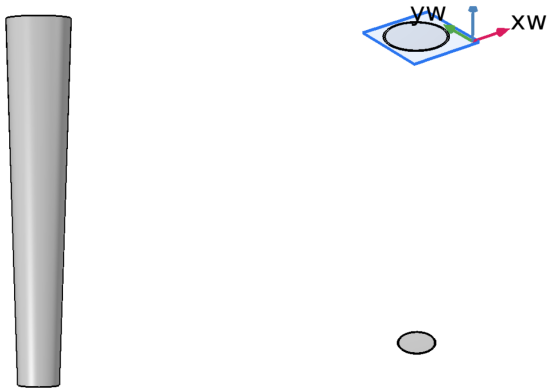


Figure 2.18. Larger fascicle due to the union of two fascicles.

2.3.5 Compute the Mesh and Solve the FEM

Once we have obtained the geometric model, we proceed to solve the FEM. To do this, as explained in the previous section, we use the `characterize_al_sites()` method, which in turn calls the `solve_single_run()` method.
At this point, we decide to pass it a file with the coordinates of the nodes where COMSOL will automatically calculate the potential. This file was generated using the `create_sites()` method in *FEMModel()* class.
This function described below:

```
def create_sites(L):

 xx = np.linspace(-L, L, 100)
 k = -1e-3
 for i in range (3):
        [x, y, z] = np.meshgrid(xx, xx, k)
        x = x.reshape((x.size, 1))
        y = y.reshape((y.size, 1))
        z = z.reshape((z.size, 1))
        site_locs = np.concatenate((x, y, z), axis=1)
        with open("node_" + str(k)+".txt", 'a+') as file:
                np.savetxt(file, site_locs)
        k = k +1e-3
  filenames = ['node_-0.001.txt', 'node_0.0.txt','node_0.001.txt' ]
  with open("nodes.txt", 'w') as outfile:
    for fname in filenames:
      with open(fname) as infile:
        for line in infile:
          outfile.write(line)
```

We used the radius of the nerve as input and 64 coordinates in order to obtain a cube of 64x64x64 dimension, this will be useful for the next phase.
The Figures 2.19. and 2.20. visually show the results obtained for the stimulation of a single Point of Sources electrode at a current of 1e-6 A through the COMSOL GUI, for the Rotating fascicles model and Splitting and Merging fascicles model.

2.4 Spinal Cord

In this section, we will address the creation of the 3D model for the spinal cord, a model useful for simulating both epidural and transcutaneous stimulations.
For the creation of this model, we followed the steps outlined in Section 2.1.1 and realized that our HMLab framework, originally designed for peripheral nerve models, could be used as a foundational framework that is easily expandable and reusable for complex models. Consequently, we utilized HMLab's existing classes and added specific ones for the spinal cord.
Once again, for this model, we made use of COMSOL and Python.

2.4.1 Architecture

To represent the spinal cord, we divided it into various components, each of which then became a class: *RootBranch()*, *SpinalRoot()*, *SpinalSegment()*, *SpinalCrossSection()* and *SpinalCord()*.
Additionally, to simulate transcutaneous stimulation, we added the Body() class for creating the biological layers that make up the back, the *Cervical_Vertebra()* class for creating the cervical vertebrae, and an *TranscutaneousElectrode()* class for a new electrode type.

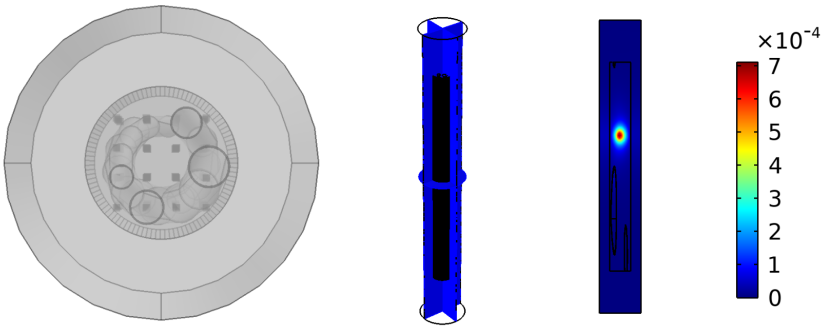


Figure 2.19. Stimulation of Rotating fascicles model.

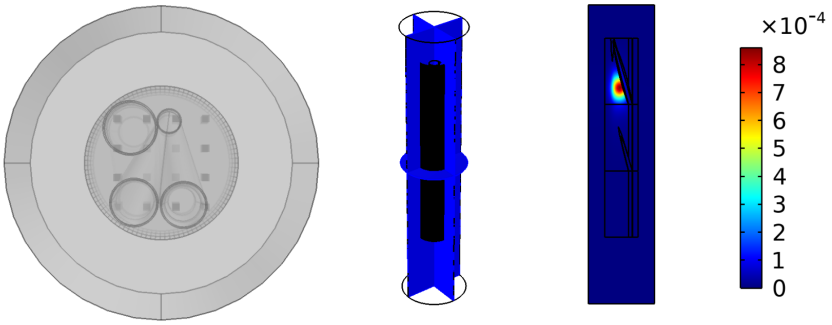
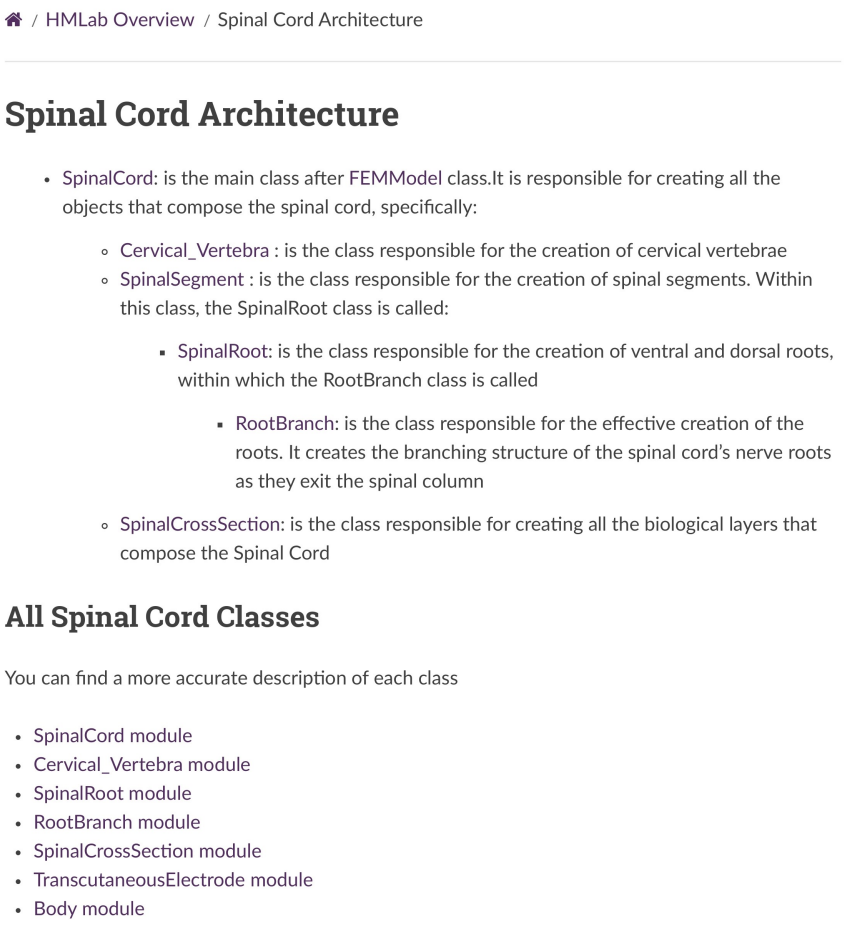


Figure 2.20. Stimulation of Splitting and Merging fascicles modes.

All of these objects, except for the electrodes, are created starting from the class called *SpinalCord()*, which is responsible for creating all the entities that compose it.
The classes that our model will utilize, which have already been used by the nerve models, are the following: *FEMModel()*, *Implant()*, *Electrode()*, and *PointSources()*. Specifically, only two types of electrodes will be used: PointSources for epidural stimulation and the TranscutaneousElectrode for transcutaneous stimulation.
Figure 2.21 is an overview of the classes dedicated to the spinal cord along with their respective functions.

Internal class base remains the same as that used for nerves, with appropriate modifications specific to different geometries.



/ HMLab Overview / Spinal Cord Architecture

Spinal Cord Architecture

- SpinalCord: is the main class after FEMModel class.It is responsible for creating all the objects that compose the spinal cord, specifically:
  - Cervical_Vertebra : is the class responsible for the creation of cervical vertebrae
  - SpinalSegment : is the class responsible for the creation of spinal segments. Within this class, the SpinalRoot class is called:
    - SpinalRoot: is the class responsible for the creation of ventral and dorsal roots, within which the RootBranch class is called
      - RootBranch: is the class responsible for the effective creation of the roots. It creates the branching structure of the spinal cord's nerve roots as they exit the spinal column
  - SpinalCrossSection: is the class responsible for creating all the biological layers that compose the Spinal Cord

All Spinal Cord Classes

You can find a more accurate description of each class

- SpinalCord module
- Cervical_Vertebra module
- SpinalRoot module
- RootBranch module
- SpinalCrossSection module
- TranscutaneousElectrode module
- Body module

Figure 2.21. Architecture of Spinal Cord Classes.

2.4.2 Model

The starting model from which we began (Figure 1) was obtained through the segmentation of images produced by MRI and CT scans on monkeys(Figure 2.22.)[**?**].
These data were then used to create the sections to which the Loft operation provided by COMSOL was applied, resulting in the monkey's spinal cord at the cervical level. This initial model was done using MATLAB, so here too as we did for nerves, we had to first perform a translation to obtain the code in Python.

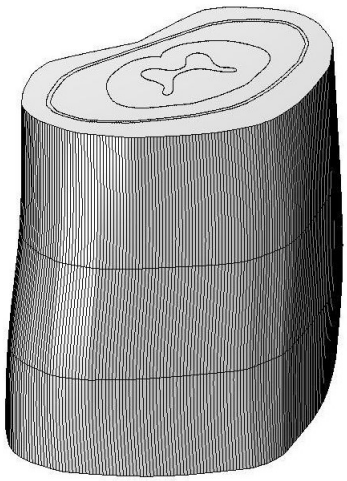


Figure 2.22. Model of Monkey Spinal Cord .

From a biological perspective, the spinal cords of primates and humans do not have significant differences except for their sizes.
Therefore, starting from this consideration, we focused on their dimensionality. In particular, we used a tool for the automatic segmentation of the human spinal cord PAM50 Template [13]to assess the difference in sizes between the spinal cords of the two species. We observed that at the cervical level the differences in the diameter length were negligible, while the discrepancy in overall height was particularly noticeable. Indeed, the length of the monkey's spinal cord is approximately 30 cm, while that of humans is about 45 cm. As a result, the first thing we did in our model was elongate the three spinal cord sections examined for the model (Figure 2.23.) [21].
Once the structure of the spinal cord was adjusted, the other entities to be constructed are:

- Spinal Root
- Vertebrae
- Body

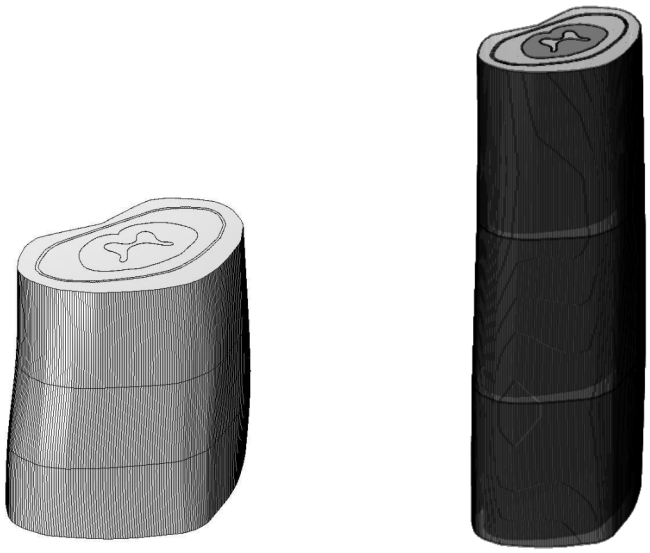


Figure 2.23. On the left Monkey Spinal Cord section. On the right Human Spinal Cord section.

Spinal Root In the original model we started from, the spinal roots were also modeled. They were present for all three modeled sections, but naturally, they were sized according to the primate spinal cord. In this case, since the project's goal is to have a more general model, we decided to temporarily neglect this part to focus on constructing other equally fundamental components.
Therefore, we retained the original root branches and adjusted their positioning along the human spinal cord as needed. The spinal cord model with spinal roots can be seen in the Figure 2.24..

Vertebrae

Measures To obtain accurate data, we conducted a search in the scientific literature to gather all the necessary measurements. Since there were no comprehensive studies containing all the required measurements for geometry construction, we selected studies that presented common and highly consistent measurements. In cases where multiple measurements were available for a given parameter, we calculated the average to obtain a more representative estimate.
The measurements taken from the scientific literature provide a crucial starting point for creating a realistic model. However, it's important to note that human vertebrae can vary significantly from individual to individual, so it's necessary to take this variability into account when developing the model.
The measurements of all the components that make up the vertebrae are visible in the

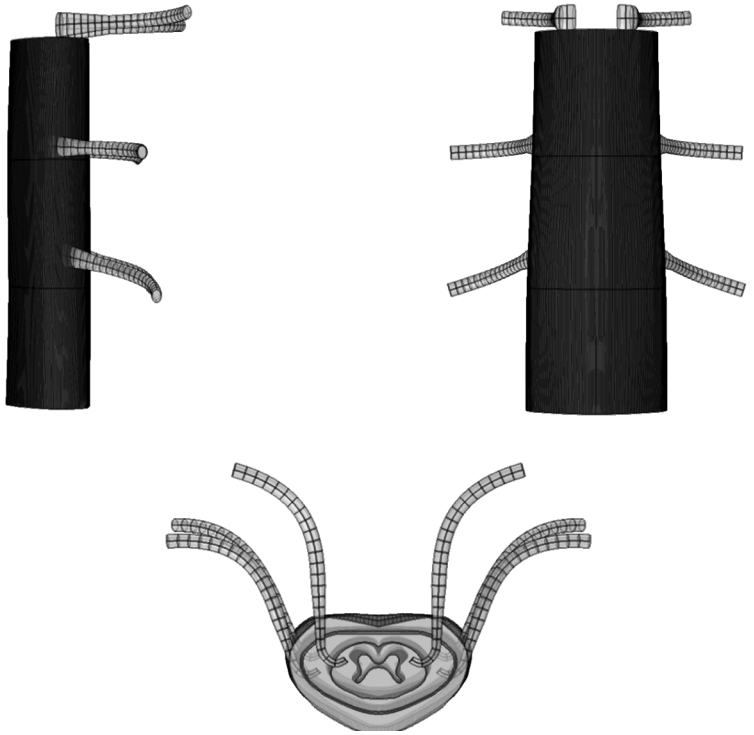


Figure 2.24. Spinal Cord model with spinal roots.

Table 2.2.[29]. However, to make the model more general and easier to implement, it was

Table 2.2. Vertebrae Measures C5, C6 e C7

Vertebra	**C5**	**C6**	**C7**
Height	13.27	13.55	14.97
Diameter Superior Endplate	16.51	17.07	17.63
Diameter Inferior Endplate	18.05	18.07	17.05
Foramen transversaria (right)	5.475	5.43	4.595
Foramen transversaria (left)	5.645	5.21	4.685
Anterior margin to anterior margin of transverse foramen	7.29	7.31	10.21
Posterior margin to posterior margin of transverse foramen	2.75	3.42	0.91
Medial margin to medial margin of transverse foramen	4.99	5.25	5.62
Lamina length	14.13	1.41	14.86
Lamina width	2.83	2.89	3.13
Lamina height	10.44	11.6	11.97
Spinal process	12.04	13.78	22.78
Total length	38.39	39.52	56.00

decided to use fixed measurements for all vertebrae in the modeled sections, C5, C6 and C7, except for the largest one, specifically C7.
All the measurements adopted for the various components are listed in the Table 2.3..

Table 2.3. Vertebral Measurements

Measurement	**Vertebra**	**Vertebra C7**
Height	13.27	13.27
Diameter Superior Endplate	14	14
Diameter Inferior Endplate	20	20
Foramen transversaria (right)	5	5
Foramen transversaria (left)	5	5
Anterior margin of vertebral body to anterior margin of transverse foramen	7.29	7.29
Posterior margin of vertebral body to posterior margin of transverse foramen	2.75	2.75
Medial margin of Luschka joint to medial margin of transverse foramen	5	5
Lamina length	14.15	14.33
Lamina width	3	3
Lamina height	9.15	9.15
Spinal process	12.12	12.75
Total length	39.35	45

We also used a 15° angle to model the typical inclination of the vertebrae [10].

Geometry To simplify the geometry and proceed with the model construction, we first analyzed the components of the vertebra. Subsequently, we assigned a geometric primitive to each of them.
Specifically, we divided the vertebra into the vertebral body, transverse foramen, spinous process, and lamina. Additionally, we also considered the biological entities present between the various vertebrae, namely: intervertebral discs, endplates, and facet joints.
Considering the state of the art and the morphology of the various components, we decided to adopt the following geometric simplifications:

- Intervertebral Discs: loft of elliptical sections [30]
- Endplates: loft of elliptical sections
- Facet Joints: loft of elliptical sections
- Vertebral Body: loft of elliptical sections
- Transverse Foramen: loft of circular sections

- Spinous Process: loft of triangular sections
- Lamina: loft of polygonal sections
- Spinal Canal : loft of elliptical sections [16]

The Figure2.25. displays the representations of these entities in COMSOL.
This is how we obtained the final model Figure 2.26. and the entire vertebrae model Figure 2.27..

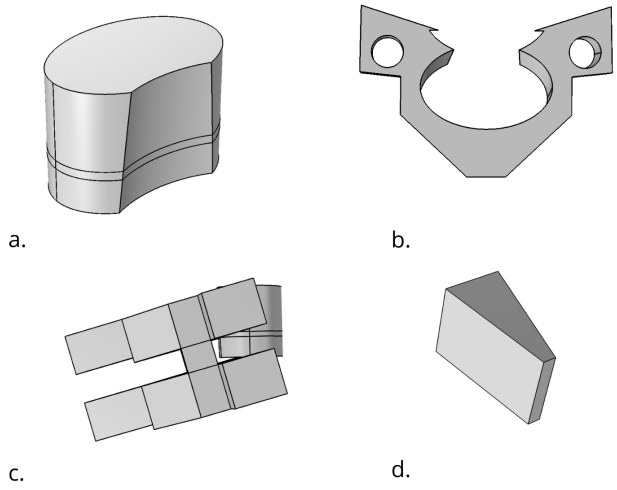


Figure 2.25. Vertebra model component. a.vertebral body with intervertebral disc and endplate, b.spinal canal and lmaina, c.facet joints, d. spinous processl

Body The back was modeled following [18] creating multiple cylinders one for each layer it consists of. Specifically, through the Body() class, starting from the outermost layer, we generated an air cylinder, which is used to create an environment with physical properties around the back. This is particularly useful for transcutaneous stimulation.
Then, a temporary metal cylinder was created for the electrode generation, specifically for transcutaneous stimulation. Moving inward, there's a cylinder representing the skin, followed by one indicating the layer of fat, until we reach the innermost layer closest to the spinal cord, defined as the general neck.
All these layers were then characterized by specific physical properties. These will be detailed in the "Materials" section of this section.
The Figure 2.28. illustrates all the different layers modeled as cylinders.

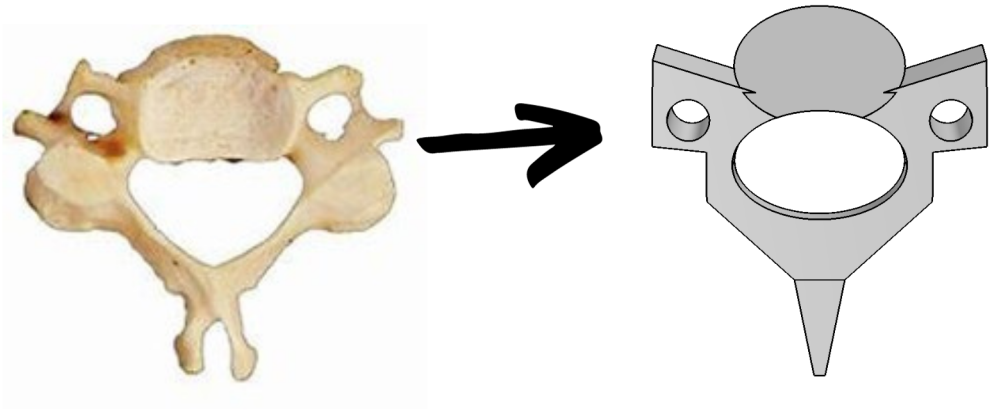


Figure 2.26. On the left a representation of an original vertebra. On the right our vertebra model

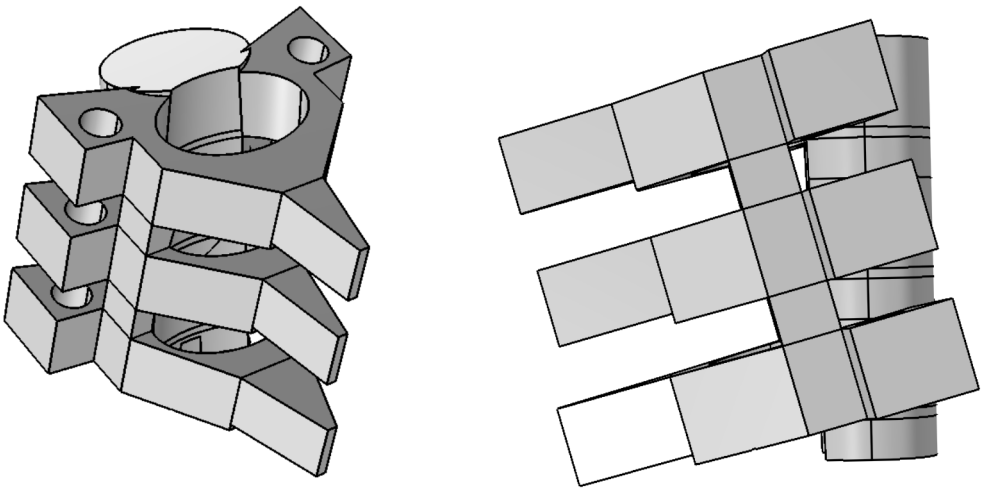


Figure 2.27. Vertebrae Model

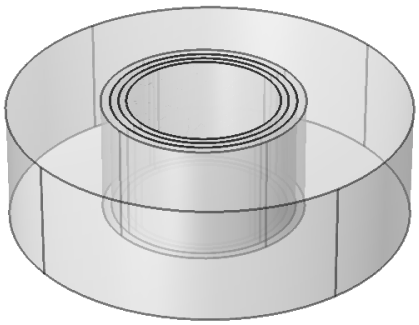


Figure 2.28. Body Model

2.4.3 Electrodes

The electrodes used for epidural stimulation are of the 'Point sources' type (see the Appendix A) and are created using the *Electrode()* and *Implant()* classes and their methods provided by HMLab. As for transcutaneous stimulation, it differs in that the electrodes do not enter the nerve but are placed on the outer layer of the skin. Therefore, we created a new type of electrode called "Transcutaneous Electrode" which has the geometry of a parallelepiped. This was created by the intersection and then the difference between the metal cylinder, an object created by the *Body()* class, and the Block object created by the *TranscutaneousElectrode()* class. Figure 2.29.shows an example of *Transcutaneous Electrode*.

2.4.4 Materials

As for the material assignment and generation process, the methods described in the previous sections were used. In this case, the materials chosen will be the only change.
We chose to assign the typical physical properties of muscles to the object representing the neck because they occupy a significant portion of the space. Since they were not yet present in the model, we assumed they could still influence the stimulation results.
The chosen materials with their properties are described in the Table 2.4. [17] [15] [28] [23] [22] [25] .

Transcutaneous Electrode

In the figure an electrode of type TRANSCUTANEOUS ELECTRODE is represented using COMSOL object

Parameters:

- width = 15
- depth = 6
- height = 15

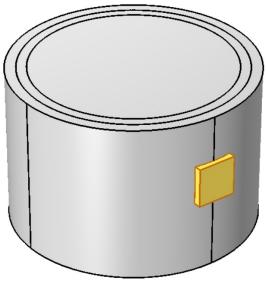


Figure 2.29. Transcutaneous Electrode.

Table 2.4. Material Properties

Tag	Name	Relpermittivity	Electricconductivity
air	Air	1.0	1.0×10^{-15}
skin	Skin	1.0×10^{-3}	0.0025
fat	Fat	1.0×10^{7}	0.04
thorax	Thorax	1	0.35
CSF	Cerebrospinal Fluid	100	1.7
fat_sc	Fat_SC	1.0×10^{7}	0.04
wm	White Matter	3.5×10^{7}	$0.083 - 0 - 0 - 0 - 0.083 - 0 - 0 - 0 - 0.6$
gm	Gray Matter	4.5×10^{7}	0.23
dura	Dura Mater	1	3×10^{-2}
bone	Bone	4.5×10^{7}	0.02
anulus	Anulus Fibrosus	4.5×10^{7}	0.75
nucleus	Nucleus Pulposus	4.5×10^{7}	0.02
cartilage	Cartilage	4.5×10^{7}	0.18

2.4.5 Simulation

In this case as well, we followed the same procedure described previously, using the same methods presented by HMLab. Obtaining for transcutaneous stimulation the model shown in Figure 2.30., where on the left is the geometric model before finite element method FEM calculation, and on the right is the model that shows the potential distribution once the FEM has been computed.

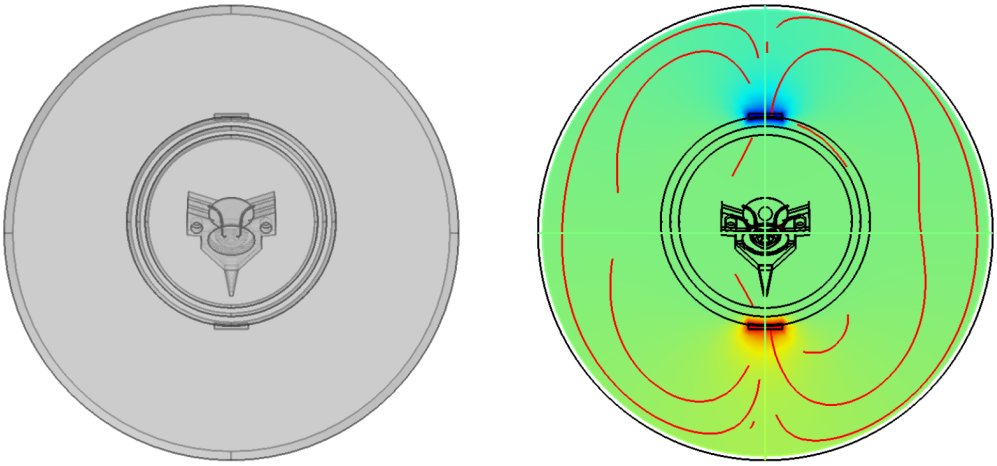


Figure 2.30. Transcutaneous Stimulation. On the left is the geometric model before FEM calculation, and on the right is the model that shows the potential distribution once the FEM has been computed.

2.5 Surrogate Model: 3D UNet

As mentioned in the introduction, HMs have a high computational cost; therefore, the use of surrogate models based on machine learning techniques is opening the way to more optimized scenarios.
So, we have decided to follow this path using a convolutional neural network (CNN) called 3D UNet [27].

2.5.1 UNet Overview

The UNet architecture is commonly used in image segmentation and medical image analysis applications.
The "U"-shaped structure with connections between descent and ascent blocks allows the model to retain detailed information during the process of reducing the spatial dimension and subsequently use it to achieve accurate segmentation.
Taking a closer look from the inside, we can identify three main steps for defining the network: Definition of DownBlocks, which is used to reduce the spatial dimension of the input, Definition of UpBlocks, which is used to increase the spatial dimension of the input, and Definition of UNet, which represents the entire architecture of the UNet.

The network we used takes two input cubes, each with dimensions of 64x64x64. The first cube represents the nerve topography, specifically indicating the presence or absence of fascicles. The second cube represents the stimulation protocol, which consists of the injected cathodic or anodic stimulation currents. The output is also a 64x64x64 cube containing the distribution of electric potential calculated for that nerve cube.
Based on this already implemented and tested scenario, our work has been divided into three phases:

1. Training the network with nerves with complex morphologies.
2. Evaluating the results of the pre-trained network for simple nerves.
3. Calculating and optimizing fiber activation.

2.5.2 Training the UNet with nerves with complex morphologies

To train the network with nerves of complex morphologies, we first analyzed the data and its format for input preparation into our network. In particular, our function required a set of MATLAB files representing the coordinates of positions in millimeters and pixels of active sites (electrodes), the potential value for each active site at each node, the nerve topography expressed in binary values, and two dataframes.
More in details it requires:

1. alpha_fem.mat : This file consists of n.electrodes x (64x64x64), where each cell corresponds to the potential value calculated for that electrode located at a specific position (x, y, z).
2. alpha_fem_<ex>_<elec>.mat: It includes two files:
   - The alpha_fem.mat
   - The alpha_fem_red: consists of 1 x (64x64x64), where each cell corresponds to the potential value calculated for that electrode <elec> located at a specific position (x, y, z) and belonging to experiment number <ex>. We have as many files of this type as there are electrodes.

3. experiment_<ex>.mat: It includes two files:
   - The nerve_solid: consists of a matrix of zeros and ones, indicating the presence of fascicles in the nerve volume.
   - The site_ind_locs: consists of n.electrodes x 3,where each row corresponds to the pixel coordinates (x, y, z) of the specific electrode. We have as many files of this type as there are electrodes.
4. site_locs: consists of n.electrodes x 3,where each row corresponds to the mm coordinates (x, y, z) of the specific electrode.
5. dataframe.pkl: you need two dataframes one for the training and one for the validation. With the following structure:
   - Id experiment
   - Number of active site
   - Id active site used
   - Value of current in mA

   See Table 2.5. for an example.

Table 2.5. Experimental Data

	id_experiment	n_active_sites	iid_active_sites	currents
0	1	1	[20]	[730]
1	1	1	[61]	[380]
2	1	1	[48]	[980]
3	1	1	[5]	[610]
4	1	1	[48]	[35]
5	1	1	[20]	[115]
6	1	1	[24]	[935]
7	1	1	[10]	[75]
8	1	1	[17]	[670]
9	1	1	[0]	[590]
10	1	1	[42]	[575]

To obtain the information necessary to create these files, we used our previously created models. Specifically, we created a nerve model with rotating fascicles to which we applied four plates with 16 Point Sources electrodes each, resulting in a total of 64 electrodes (see the COMSOL model in the Figure 2.31.).
Subsequently, using the methods from the FEMModel() class explained in the first section, we calculated the FEM for each active site in 64 nodes, which remained the same for each site.
This process yielded 64 lead field matrices of size 64x64x64.

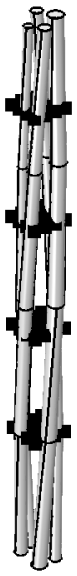


Figure 2.31. Nerve model with rotating fascicles to with four plates with 16 Point Sources electrodes each.

This information is not sufficient. To obtain the files with the coordinates of our active sites, we added two functions, called in PointSources() class in HMLab. One function, `save_coordinates()`, is used to save the positions in millimeters of our sites and to call another function, `mm_pixel()`, used to convert these positions from millimeters to pixels and then save them. `save_coordinates()` takes as input the coordinates of the active site and the minimum section length represents one pixel.
In particular, we adopted the following conversion strategy:
Considering our cube with dimensions 64x64x64, we performed the following steps:

1. Assigned the value 32 to $x = 0$ and $y = 0$ mm since it is the average value within the cube. This step was necessary because pixels cannot have negative values.
2. Assigned the following values to all other x and y values:
   - $y = 32 + y * 32$
   - $x = 32 + x * 32$

   This was done to ensure non-negative pixel values.
3. Divided the total length of the nerve by 64 to obtain the length of nerve sections corresponding to one pixel.
4. Assigned $z = (z\_mm/section\_length)$.
5. Rounded x, y, and z to the nearest integer since pixels accept integer values.

To create the datasets to use as the training set and validation set, we used the `build_dataframe()` function, which creates a dataframes structured as shown in the example before.
The last piece of information needed to proceed with creating the files for training is the binary representation of the nerve's topography.
This step is where we encountered the most difficulty due to the complex geometry of the fascicles. The matrix we aimed for consists of 0s and 1s, where 1 indicates the presence of a fascicle and 0 indicates its absence. By "matrix," we are referring to our 64x64x64 pixel cube representing our nerve.
As a result, we decided to divide this task into two sub-tasks:

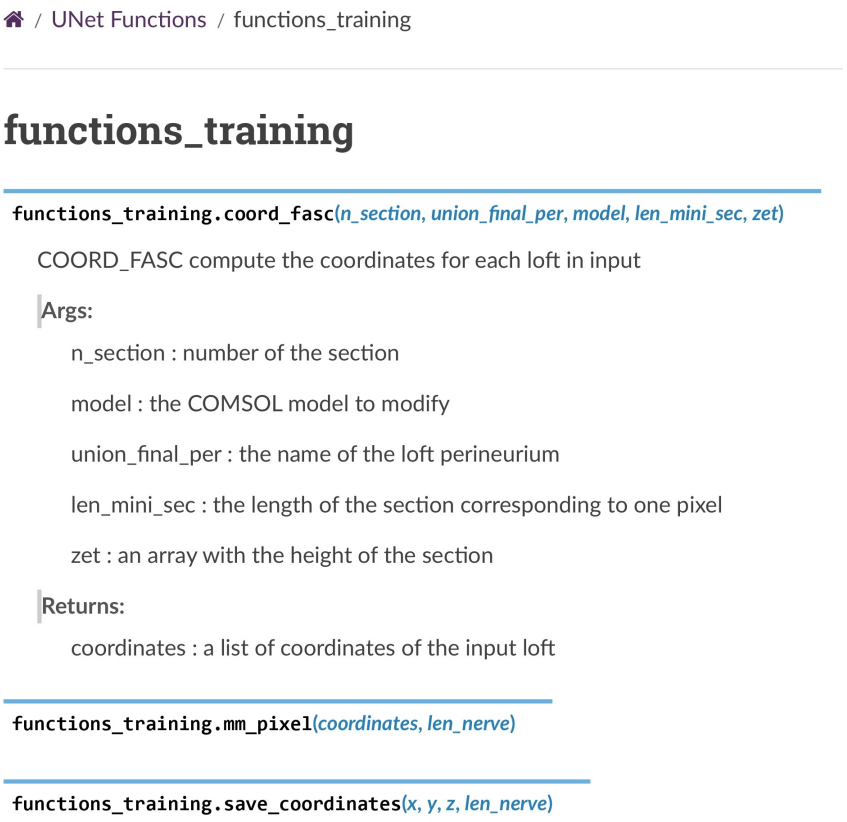
1. Calculating the coordinates of the fascicles for each section.

2. Converting the coordinates calculated in step 1 into pixels and creating the cube with 0s and 1s.

Calculating the coordinates of the fascicles for each section To accomplish this, we created a function called `coord_fasc()`, which takes as input the number of the section that composes the nerve (see nerve model construction with complex morphologies), the name of the loft indicating the perineurium of a fascicle for that section, model, an array with the height of that section, and the length of the section corresponding to one pixel, obtained by dividing the length of the nerve by 64.
Figure 2.32. provides a clearer outline of the function's structure.

Within this function, through a for loop iterating as many times as there are pixels in the section, we create small COMSOL cylinders and calculate their intersection with the loft. Here, thanks to the use of the COMSOL method:
`model.java.component(<comptag>).geom(<geomtag>).measure().getVtxCoord()` [8]
, we are able to obtain the x, y, z coordinates of the cylinder-fascicle intersection. We perform these operations for all the mini-sections representing one pixel in the nerve section, and we obtain a list with the intersections coordinates.
The `coord_fasc()` function is called during the creation of each fascicle in the model, in the `add_to_model() method`.

All of these steps became necessary because when we create the rotating loft or the loft that splits or merges, we don't have precise control over the exact positions it will occupy along the path between the input sections defined for the loft.



/ UNet Functions / functions_training

functions_training

functions_training.coord_fasc(*n_section, union_final_per, model, len_mini_sec, zet*)

COORD_FASC compute the coordinates for each loft in input

Args:

n_section : number of the section

model : the COMSOL model to modify

union_final_per : the name of the loft perineurium

len_mini_sec : the length of the section corresponding to one pixel

zet : an array with the height of the section

Returns:

coordinates : a list of coordinates of the input loft

functions_training.mm_pixel(*coordinates, len_nerve*)

functions_training.save_coordinates(*x, y, z, len_nerve*)

Figure 2.32. coord_fasc function.

Converting the coordinates calculated in step 1 into pixels and creating the cube with 0s and 1s Once we have all the intersections coordinates of all the fascicles in all the mini-sections of the nerve section under consideration, we proceed with the conversion to pixels and populating our binary cube.
To do this, we created another function called `pixel_topography()`. Starting from the list with all the intersection coordinates for each fascicle and for each section-pixel, it selects only the maximum and minimum points, only relevant coordinates, and calculates the radius and center using our created function `find_circle_center_and_radius()`.
Once this is done, using another function we created, `points_on_circle()`, we calculate n_points that belong to the circumference defined by the previously calculated radius and center. Then, we call the `mm_to_pixel()` function, which, invoking `mm_pixel()` function, converts the coordinates of all the points belonging to the circumference into pixels and set the cells, corresponding to the pixel coordinates, of the final cube to 1.
To conclude, we populate our 64x64x64 matrix, setting all the pixels included in the circumference we defined to 1. And then we save this final cube.
We repeat this process along the entire nerve.

After completing these tasks, we have all the information to create the training files described earlier. To do this, we have created a script called `create_file_training.py` that contains all the functions dedicated to this purpose.
For greater clarity, Figure 2.33. presents the structure of all these functions.

Figure 2.33. create_file_training.py script..

At this point, everything is ready to proceed with the training.
For clarity, let's outline all the steps followed to train the model and obtain a visual representation of the results.

1. Define training dataset: download the dataframe created before.
   With `CustomDataset3d()` starting from the training dataframe and all the the training files, you can define the training dataset.

```
mode = "potential"
dataset_training = CustomDataset3d(
    experiment_info_path="E:\\surrogate_fem\\3d_data\\dataframe_training_01.pkl",
    data_folder_path="E:\\surrogate_fem\\3d_data\\"
 )
```

2. Define validation dataset: download the dataframe created before.
   With `CustomDataset3d()` starting from the training dataframe and all the the training files, you can define the validation dataset.

```
dataset_validation = CustomDataset3d(
   experiment_info_path="E:\\surrogate_fem\\3d_data\\dataframe_validation_02.pkl",
   data_folder_path="E:\\surrogate_fem\\3d_data\\"
 )
```

3. Define DataLoader and criterion: `DataLoader()` is a PyTorch [4] function. With `compute_custom_loss()` you compute a custom loss for comparing predicted and target output.

```
dataloader_training = DataLoader(dataset_training, batch_size=1, shuffle=True)
dataloader_validation = DataLoader(dataset_validation, batch_size=1, shuffle=False)
criterion = compute_custom_loss
criterion_parameters = dict(loss_func="MSE",restrict_to_fascicles=False)
```

4. Define Trainer() : With Trainer() class you initializes a training and validation manager for machine learning models.

```
trainer = Trainer(
   model=model,
   device=device,
   criterion=criterion,
   optimizer=optimizer,
   training_data_loader=dataloader_training,
   validation_data_loader=dataloader_validation,
   n_epochs=1,
   criterion_parameters=criterion_parameters,
   validation_inverse_log=False
   )
```

5. Perform Training: With `trainer.run_trainer()` you run the training and validation loops for the specified number of epochs.

```
training_losses, validation_losses, validation_losses_overall_MSE,
validation_losses_fascicles_MSE, validation_losses_overall_MAE,
validation_losses_fascicles_MAE, validation_losses_overall_Huber,
validation_losses_fascicles_Huber = trainer.run_trainer()
```

You can make prediction and plot it following these steps:

1. Make Prediction

```
torch.manual_seed(0)
dataloader_validation = DataLoader(dataset_validation, batch_size=2, shuffle=False)
                        sample_in, sample_out = next(iter(dataloader_validation))

model.eval()
device = torch.device('cpu')
model.to(device)
pred_out = model(sample_in)

sample_in = sample_in.cpu()
sample_out = sample_out.cpu()
pred_out = pred_out.detach().cpu()
```

2. Plot prediction: With `plot_data_3d()` you can plot 3D data slices including input, target, and predicted data.

```
plot_data_3d(
input_data = sample_in,
target_data = sample_out,
predicted_data = pred_out,
figsize = (25, 100)
)
```

Figure 2.34. shows a prediction example for nerve with simple morphologies, in the first column you can see the nerve topography, in the third the target output and in the last the predicted output.
The same procedure was used for the model of nerves with fascicles that split and merged, and it can be used for every model.

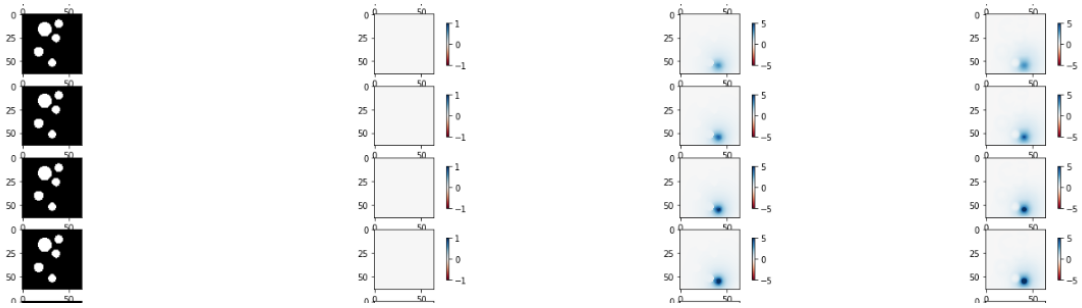


Figure 2.34. Prediction for nerve with simple morphologies, in the first column you can see the nerve topography, in the third the target output and in the last the predicted output.

2.5.3 Evaluating the results of the pre-trained UNet for simple nerves

Starting from a pretrained model for nerves with straight fascicles, we have worked on creating functions capable of making the prediction results more understandable.
We have, therefore, created a script containing a series of functions dedicated solely to plotting various graphs. To do this, we have also used functions to calculate various types of errors, including:

- Huber Loss: Huber Loss is an error function that combines the behavior of Mean Absolute Error (MAE) and Mean Squared Error (MSE). It is less sensitive to outliers compared to MSE and is useful when a robust error metric is needed.

$$L_H(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \tag{2.1}$$

- Mean Squared Error (MSE): MSE calculates the average of the squared differences between predicted values and actual values. It is widely used as an error metric for regression tasks and penalizes large errors strongly.

$$MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.2}$$

- Mean Absolute Error (MAE): MAE calculates the average of the absolute differences between predicted values and actual values. It is less sensitive to outliers compared to MSE and is more interpretable.

$$MAE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.3}$$

- Mean Absolute Percentage Error (MAG): MAG measures the average percentage error between predicted values and actual values. It is useful when evaluating error

relative to the data scale.

$$MAG(y) = \frac{1}{n-1}\sum_{i=1}^{n-1}|y_{i+1} - y_i| \tag{2.4}$$

- Correlation Coefficient (CC): CC measures the linear correlation between model predictions and actual values. The value ranges from -1 (perfect negative correlation) to 1 (perfect positive correlation).

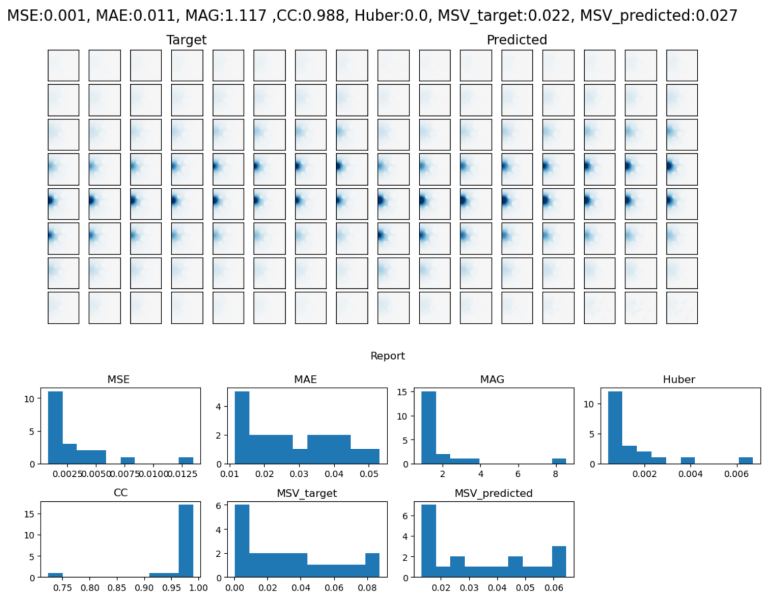
$$CC(y,\hat{y}) = \frac{\sum_{i=1}^{n}(y_i - \bar{y})(\hat{y}_i - \bar{\hat{y}})}{\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}\sqrt{\sum_{i=1}^{n}(\hat{y}_i - \bar{\hat{y}})^2}} \tag{2.5}$$

- Mean Squared Variance (MSV):MSV is a statistical metric used to measure the average squared deviation of a set of values from their mean (average).

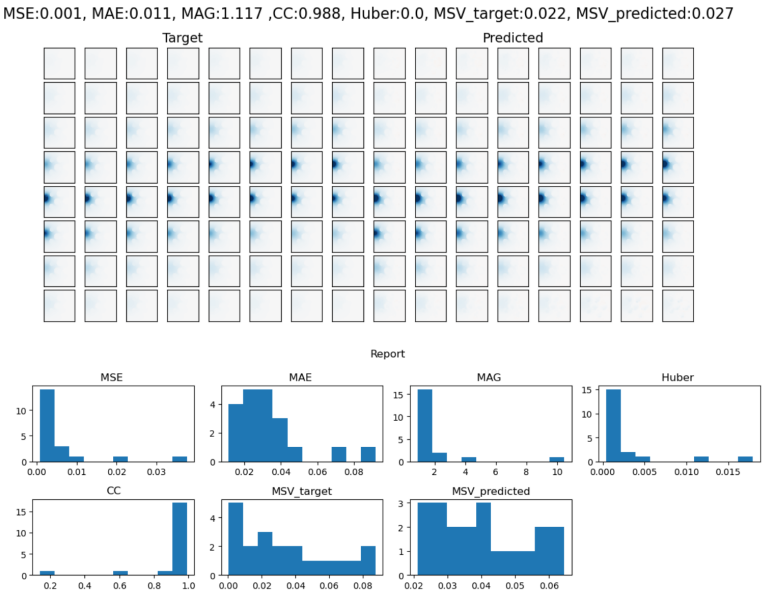
$$MSV = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2 \tag{2.6}$$

Now let's present the graphs resulting from the use of these functions. For all these plots we considered one active site at the time.

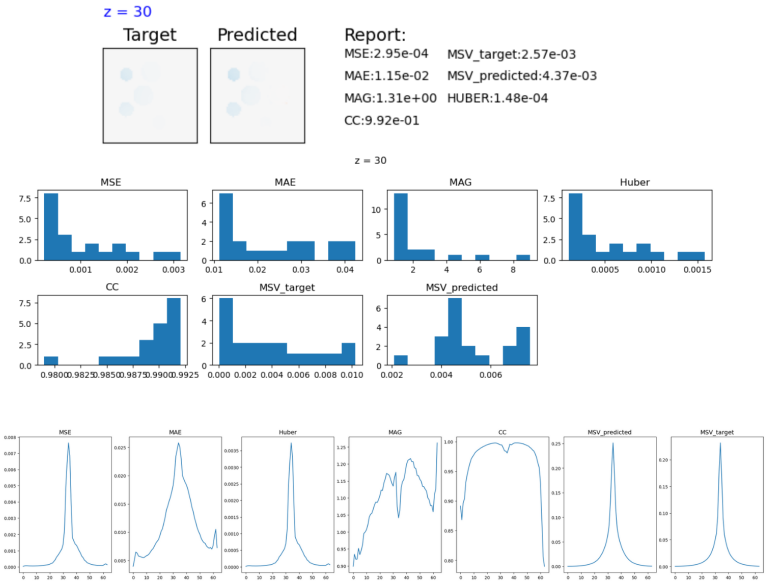
1. Report of prediction of n_samples for the same active site with different currents, along with histograms of errors. Plot single sample.



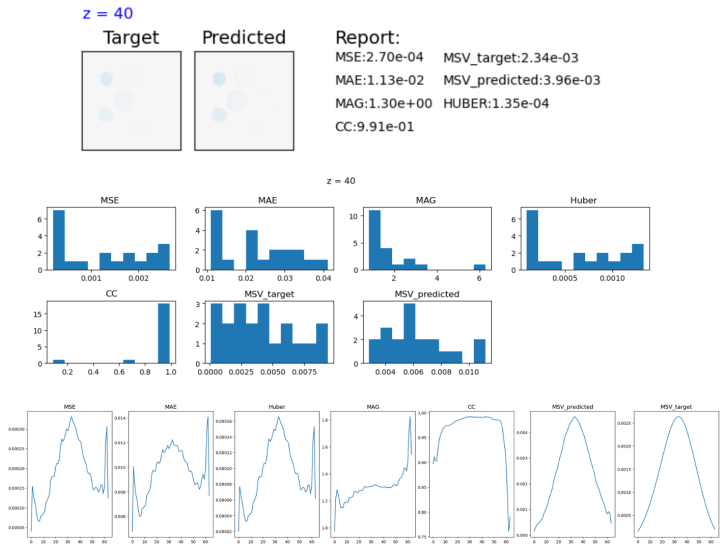
2. Report of prediction of n_samples for different active sites with same current, along with histograms of errors. Plot single sample.



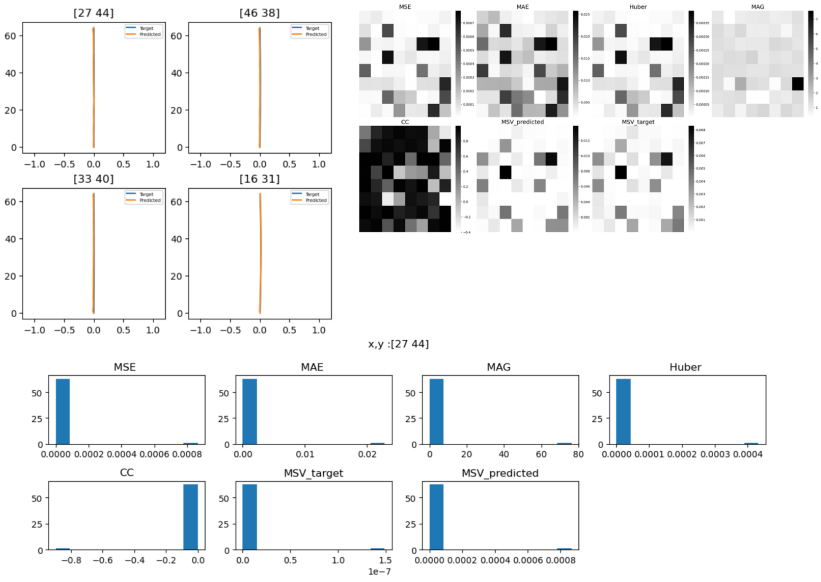
3. Report prediction at each Z levels same active site different currents, along with errors plots. Plot single sample.



4. Reportprediction at each Z levels different active sites same currents along with errors plots.Plot single sample.



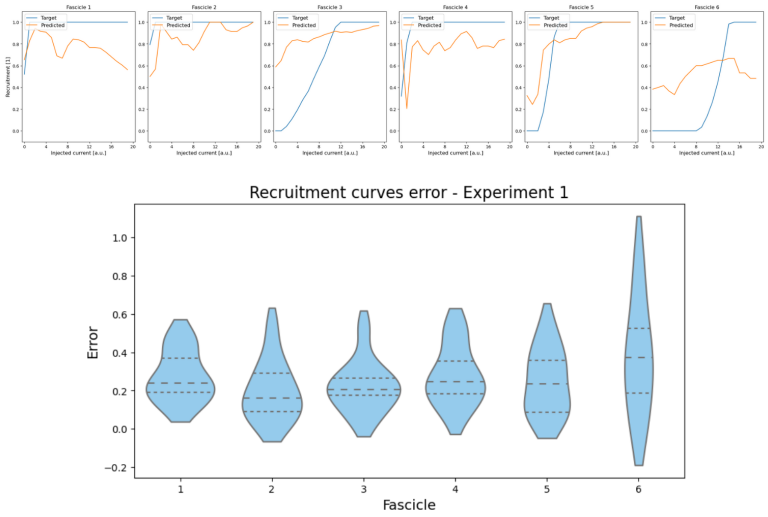
5. Report prediction at random x, y values for same active site different currents, along with along with errors plots and heat map.
   Plot single sample.



6. Recruitment: In this context "recruitment" is used to assess the activation of fascicles within a neural simulation. The code calculates how many sites within a fascicle are activated in response to a given stimulus. This information is important for understanding how different parts of the neural network respond to specific inputs

and can be used to evaluate the performance of neural models. Here the recruitment is computed in this way:
Within each fascicle, the function calculates recruitment metrics:

- The area of the fascicle in the target output that exceeds a threshold (indicating recruitment).
- The area of the fascicle in the predicted output that exceeds the same threshold.
- The recruitment metric for the fascicle in the target output, calculated as the ratio of recruited area to total fascicle area.
- The recruitment metric for the fascicle in the predicted output, calculated similarly.



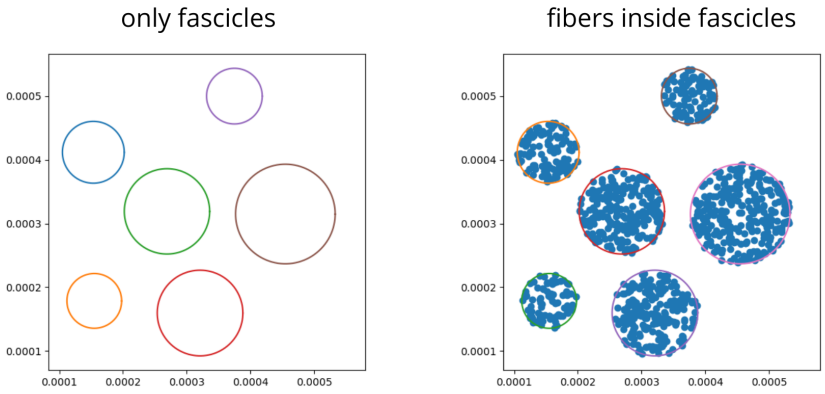
2.5.4 Calculating and optimizing fiber activation

In this section, we illustrate the work done with the fibers inside the fascicles. Here too, we used the pretrained network for nerves with simple morphologies.
Before anything else, we generated the fibers inside each fascicle. To do this, we:

1. Starting from the input cube, we extracted the x and y coordinates of the fascicles' centers and their radii using the `regionprops_table` and label functions from skimage.measure [7].

2. We generated fibers inside each fascicle using the `sample_homo_in_polyshapes(polyshapes, n_samples)` function. This function generates random points inside various polygonal shapes based on their relative areas, ensuring that the points are evenly distributed within each polygonal shape. We then calculated the diameter of the fibers

based on a uniform random distribution within a specified range.

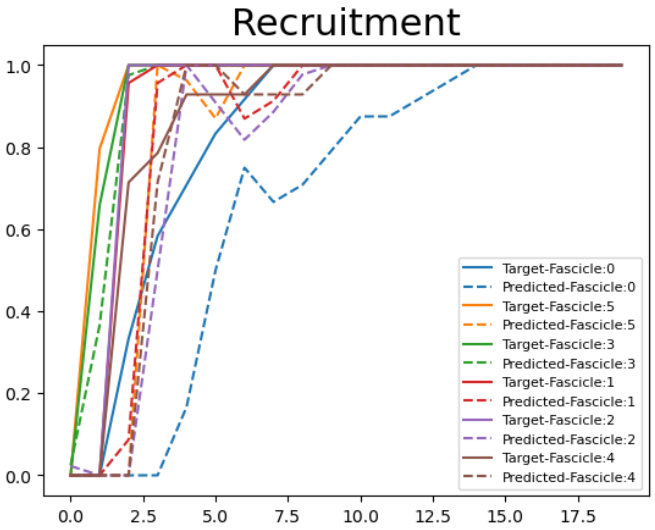


Then we can compute the activation of each fiber at a given current:

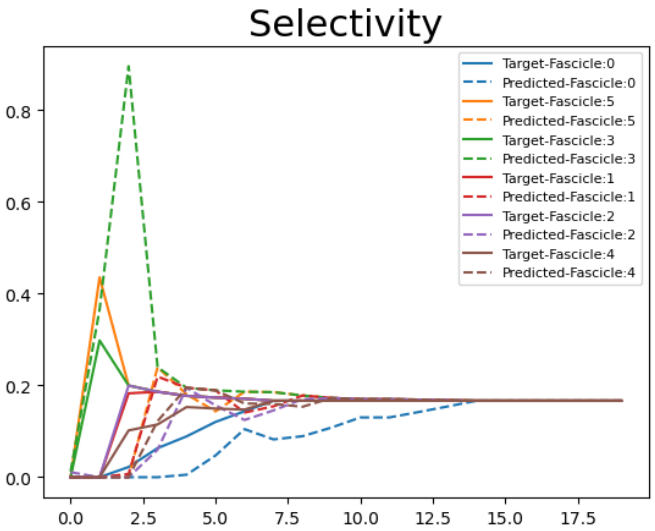
1. First of all we extract the x and y positions and calculate the z positions. Afterward, by iterating through the various current configurations and using an interpolator, we obtain the target and predicted values for specific positions along the current fiber and current configuration. These values are used to populate two datasets.

2. Afterward, we used a binary classifier based on XGBoost [6] (XGBClassifier) to calculate accuracy on both the "target" dataset and the "predicted" dataset. XGBoost is a widely used machine learning algorithm for both classification and regression problems. It is based on the sequential construction of weak decision trees and their subsequent aggregation to obtain a more robust and accurate prediction model. This technique is known as "gradient boosting."
   In both cases, we obtained an array of the same length as the number of fibers, consisting of binary values (0 or 1), where 0 indicates that the fiber was not activated by that stimulation protocol, and 1 indicates that the fiber was activated by the corresponding stimulation protocol.

3. We subsequently calculated recruitment for each fascicle as the number of active fibers for that fascicle divided by the total number of fibers in that fascicle.



4. Finally, we calculate selectivity as the square of the recruitment value for a fascicle divided by the sum of the recruitments of all the fascicles.



All the figures shown above refer to the stimulation of a single active site. These same procedures can be followed for stimulation with multiple sites. In this case, Table. shows the dataset will be generated, Figure 2.35. shows surfaces for recruitment and Figure 2.36. the selectivity will be obtained.
For all these computation specific functions were created and documented in HMLab Tutorial.

Table 2.6. Experimental Data

	id_experiment	n_active_sites	iid_active_sites	currents
0	1	2	[20; 92]	[0.0; 0.0]
1	1	2	[20; 92]	[0.1111; 0.0]
2	1	2	[20; 92]	[0.5556; 0.2222]
3	1	2	[44; 70]	[0.0; 0.0]
4	1	2	[44; 70]	[0.1111; 0.0]
5	1	2	[44; 70]	[0.5556; 0.2222]

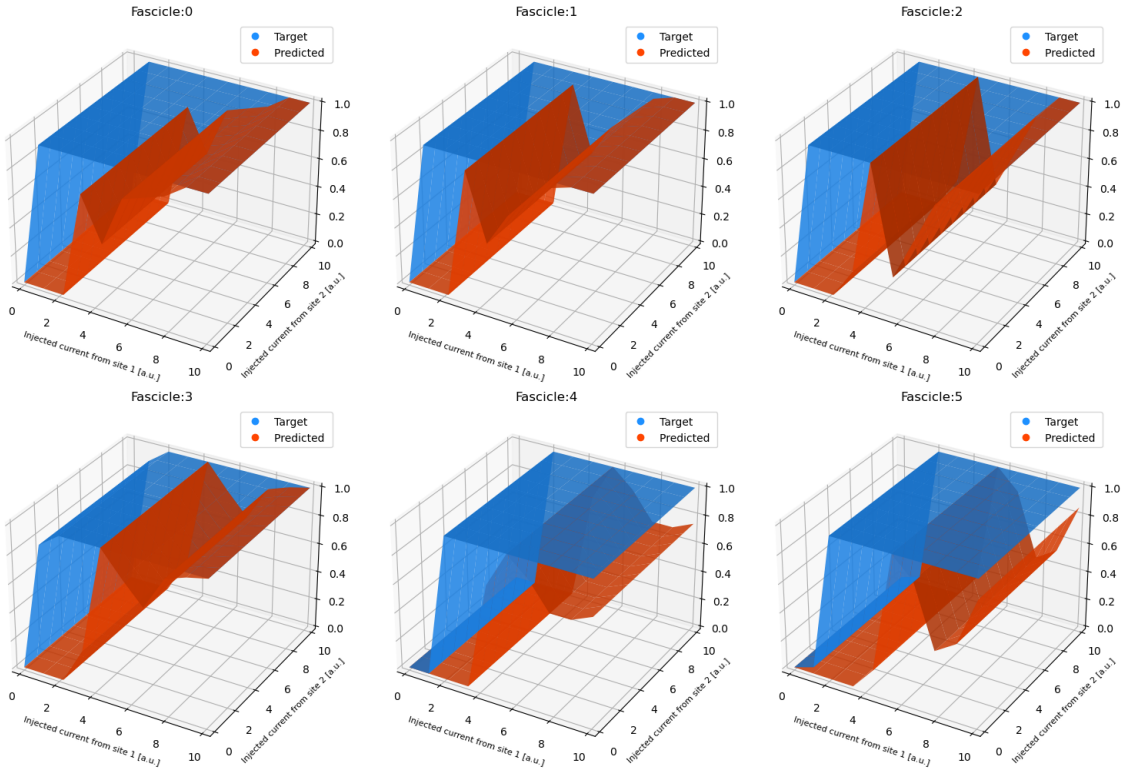


Figure 2.35. Recruitment of stimulation multiple sites.

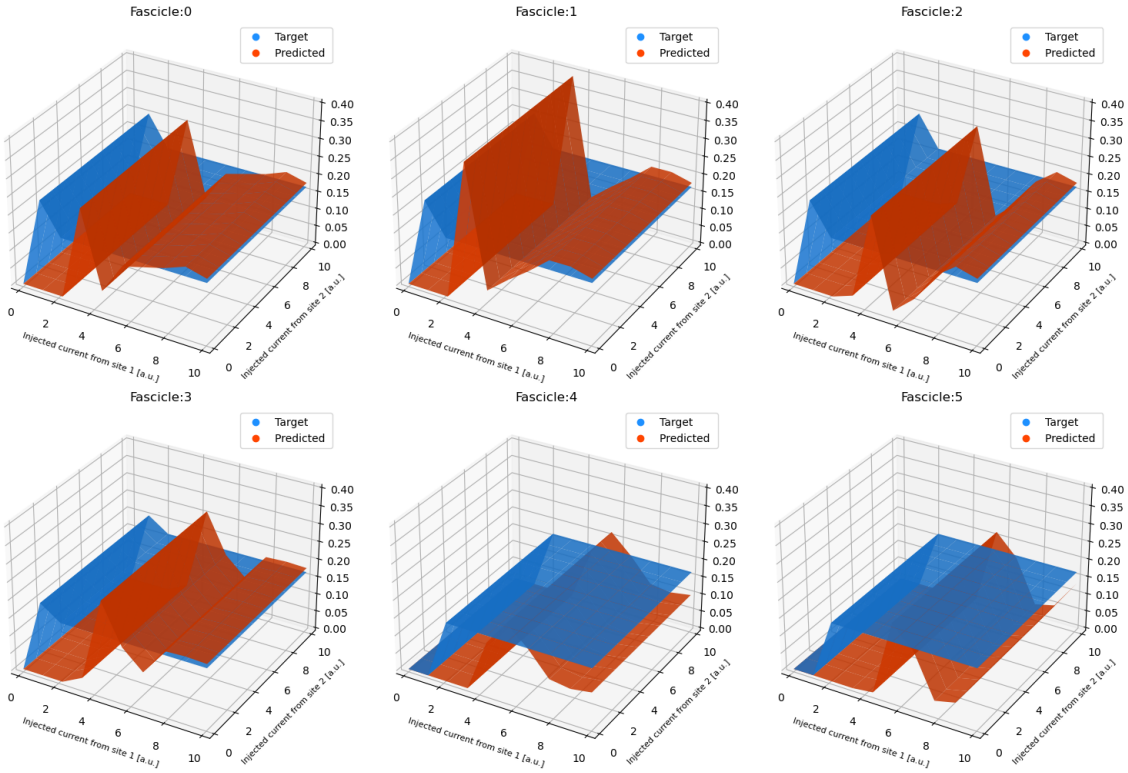


Figure 2.36. Selectivity of stimulation multiple sites.

Part III

Chapter3

Chapter 3

Discussions

3.1 Results

In the first part of our thesis, our goal was to demonstrate the ease of construction of conventional HMs for any biophysical entity using standard and automated procedures. In the second part, we focused on the partial replacement of traditional HMs by surrogate models based on machine learning, in particular we used 3D UNet.

The first phase resulted in the following outcomes:

1. We developed the HMLab framework for simple nerve geometries.
2. We expanded HMLab and developed models for complex nerve geometries.
3. We further expanded HMLab and implemented models for epidural and transcutaneous spinal cord stimulation.

First, we developed a Python framework called HMLab, which effectively exploits the technical features of object-oriented programming languages, in particular modularity. As a result, we were able to generate various models for peripheral nerve stimulation with straightforward morphologies (straight fascicles) in an automated manner. HMLab enables the production of basic models that can be personalized by selecting options such as electrode type, position and dimensions, fascicle type, and numerous other parameters.

We have further expanded the range of possible models that can be created using HMLab. This was achieved through a comprehensive study of nerve morphology, which we found to be highly variable, especially at the fascicle level. In fact, fascicles do not always follow a straight path, but can rotate, diverge, join, and intertwine. We concentrated on examining fascicles that rotate and those that split and later unite. Specifically, we generated peripheral nerve models with fascicles rotating but not intersecting, and with fascicles that split and then join with other fascicles. In addition, we demonstrated how the modularity of HMLab facilitated the implementation of these additional geometries through the addition of specific classes and methods. HMLab enables the use of pre-existing classes

designed for simple models to also characterize more complex models. The models presented in this thesis involve the utilization of Point Source type electrodes.

In the final phase of the initial stage, we examined the essential components required to depict spinal cord stimulation, including the spinal tract and surrounding environment. Following this, we developed the model for epidural and transcutaneous stimulation. During this process, we utilized HMLab, incorporating existing base classes and implementing new ones relevant to the components of this updated geometry.

The results of the second phase can be summarized in these points:

1. Analysis of predictions from the network already trained on simple nerves.

2. Preparation of training data for nerves with complex morphologies.

3. Generation of fibers within the bundles and calculation of their activation.

First, building upon the pre-existing network's exposure to elementary nerve structures, our goal was to create functions that would allow us to better understand the results. Through meticulous evaluation of errors and the production of various plots, we thoroughly examined the accuracy of our predictions.

Subsequently, a procedure was developed to translate the output data and geometries generated by traditional HM into input data for our network. We transformed the coordinates measured in millimeters of the hybrid model into pixel coordinates, which led to 64X64X64 pixel cubes. Next, we depicted the topography of the fascicles in a binary matrix, indicating their presence or absence. We trained the network, but with a limited dataset, significant results have not been obtained yet.

Finally, fibers were produced within the fascicle s and their activation was calculated through specific stimulation protocols using a binary classifier. The data obtained allowed us to compute the recruitment and selectivity of each fascicle in relation to different stimulation. This selectivity is important in neuroprosthetics applications since it enables us to precisely target and activate a specific region, such as a particular muscle.

Our framework has been thoroughly documented and use cases and tutorials have been provided, to maximize the usability and expandability of the framework, both through the inclusion of further modelling modules and machine learning surrogate models.

In cocnclusion,we have illustrated the ability of our framework, developed by utilizing Python's modularity features and COMSOL software, to generate a variety of geometries automatically and intuitively, without the necessity of manual segmentation or specific CAD software. This has the potential to lay the foundation for any kind of biophysical model.

3.2 Limitations of the Study and Future Investigations

Through HMLab we have defined a method to automatically and programmatically generate different types of models such as models of transcutaneous spinal cord stimulation and and elaborate fascicular morphologies in peripheral nerves.

Traditionally, previous models only focused on the spinal cord without considering the impact of the surrounding bones during transcutaneous stimulation. Our goal was to establish a foundation for a more comprehensive model of the cervical region that encompasses not only the cervical spine tract, but also the environment around it.
The presented model is in its early stages. In particular, the model lacks the muscles that represent with their biophysical properties a relevant component in a context of spinal stimulation. Additionally, modeling root branches with anisotropy, which involves imposing proper current directionality, may lead to enhanced accuracy and reliability in simulations.
Clearly, the ideal approach would be to create a personalized model for each patient, based on individual MRI data. However, to gain preliminary insights into the optimal current pathway for transcutaneous stimulation, we have opted to begin with a model based on aggregated data from scientific literature, related to average subjects.

We also modeled rotating fascicles and merging and splitting fascicles. Fascicles in a nerve are groups of nerve fibers gathered. The size of fascicles is an anatomical characteristic that can vary greatly among individuals and different nerves, making it challenging to maintain an accurate understanding of their topography. Therefore, we deemed it necessary to develop stimulation models that portray more intricate topographies than those previously presented in the literature. To create all these models, we followed the steps outlined in and detailed in HMLab, using COMSOL Multiphysics to solve the volume conduction problem and create 3D models. The utilization of COMSOL enabled us to automatically generate the mesh and compute the electrical potential at each node. The generation of complex geometries brought to light the difficulty of constructing the mesh and arriving at study convergence. The spatial arrangement and size of individual geometry elements affect mesh resolution and the ability to obtain realistic results.

Instead, the complexity of the geometry has no impact on the quality of surrogate model results. Surrogate models substitute traditional HMs' model construction, which is based on geometric primitives, with machine learning techniques that allow the prediction of the resulting electrical potential. In particular, we introduced the 3D UNet , a convolutional neural network mainly employed for biomedical image segmentation, which uses two input cubes, each with a dimension of 64x64x64, to pursue this goal.
So, we have demonstrated the potential of surrogate models to streamline the process. However, to take advantage of this accelerated approach and achieve accurate predictions, it is crucial to automatically generate a multitude of plausible geometries. We also started to build the input dataset of the network in order to train it with complex nerve

morphologies, but to train the network on these geometries, they must be produced using the pipeline of traditional models, which may be affected by the complexity of the geometry to solve the mesh and obtain reliable results.
Additionally, UNet has a structural limitation as the evaluation of the solution occurs within a regular cubic grid. This choice is suboptimal when dealing with complex geometries, especially if composed by elements with divergent physical properties. For instance, when examining the electroconductivity of a peripheral nerve's fascicle, it consists of the endoneurium with an electroconductivity level of 0.083 and the perineurium with an electroconductivity level of 0.0009. Given that the perineurium is a thin layer around the endoneurium, the dissimilarity in their biophysical attributes could be lost during the discretization in the 64x64x64 matrix required by UNet.
Furthermore, representing the full nerve topography in a 64x64x64 binary cube would result in errors due to dimensionality reduction. This could generate topographies with fused fascicles where there is actually a discontinuity present. As a result, UNet remains unaware and produces inaccurate predictions. While it is true that traditional HMs are computationally expensive, one should not underestimate the significant number of resources required to properly run the UNet. This includes handling all the network weights, which requires a considerable amount of memory.
Another potential advantage offered by the use of surrogate models is the ability to implement optimization algorithms capable of exploring the entire solution space to identify the best desired stimulation protocol. These algorithms can be based on functions already present in the literature, such as the "Particle Swarm" algorithm, to predict, given a current value, the optimal location of an active site to stimulate a specific nerve area or to predict the best current value to use, ultimately leading to the automatic prediction of the most suitable stimulation protocol for a given area to be stimulated.

Part IV

Chapter4

Chapter 4

Conclusion

We have demonstrated how our framework can automatically and programmatically generate a wide range of models. By combining methods from abstract classes and creating special ones for specific cases, we have demonstrated how a limited set of geometric primitives can model various areas of the body, including nerves, spinal cords, and both invasive and non-invasive stimulation. We have also shown how surrogate models can replace these primitives to speed up the process. However, generating numerous reasonable geometries automatically is essential to leverage this speed and to obtain correct predictions. Yet, complex geometrical elements pose a challenge for meshing. Among surrogate models, we introduced the 3D-UNet. This model has a structural limitation as the evaluation of the solution occurs within a regular cubic grid. This choice is suboptimal when dealing with elaborate geometries, especially if composed by elements with divergent physical properties. Our work paves the way for the automatic generation of any biophysical model. However, additional enhancements are necessary to address challenges arising from complex meshes, enabling us to use it on a larger scale. Improving the automatic creation of these models would increase the dataset used to train UNet, allowing for the exploitation of its full potential. Additionally, the high speed of prediction enables the generation of optimization algorithms that can quickly and automatically explore the entire parameter space and choose the most suitable stimulation protocols for various needs.

Part V

Appendix

Appendix A

Electrodes

Electrodes Types

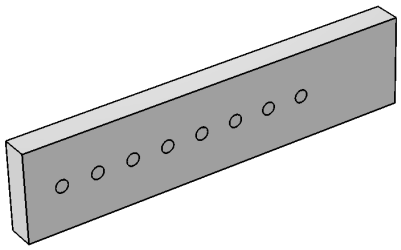
The HMLab allows you to choose among different types of electrodes:

Time & Grounded Time

In the figure an electrode of type TIME is represented using COMSOL object

Parameters:

- length of the electrode shaft = 1.5 mm
- width of the electrode shaft = 0.5 mm
- height/depth of the electrode shaft = 0.15 mm
- distance between same face active sites = 0.2 mm
- number of active sites per face = 8
- diameter of active sites = 0.07 mm
- depth of active sites = 0.01 mm
- origin location of the electrode reference frame:
  - x = 0
  - y = 0
  - z = 0
- orientation of the electrode referenc frame:
  - theta_x = 0
  - theta_y = 0
  - theta_z = 0



Cuff

In the figure an electrode of type CUFF is represented using COMSOL object

Parameters:

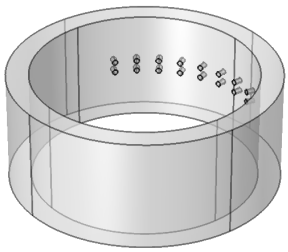
- radius of the electrode = 1 mm
- external radius of the nerve = 1.25 mm
- length of the electrode = 5 mm
- thickness of the electrode = 0.1 mm
- distance between active sites = 2.5 mm
- length of active site = 2 mm
- depth of active site = 0.05 mm
- width of active site = 0.25 mm
- center-to-center distance between sites = 0.7
- z-position of active site = 0
- angle of insertion around z axis = 0

Soft Cuff

In the figure an electrode of type SOFT CUFF is represented using COMSOL object

Parameters:

- radius of the nerve = 1 mm
- length of cuff along the nerve = 1 mm
- thickness of the cuff = 0.2 mm
- active site diameters = 0.05 mm
- distance between active sites = 0.2
- number of active sites = 8 mm
- active site depth = 0.07 mm
- distance between active site rings = 0.1 mm
- z-position of active site = 0
- angle of insertion around z axis = 0



Cylinder Cuff

In the figure an electrode of type CYLINDER CUFF is represented using COMSOL object

Parameters:

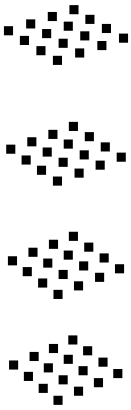
- radius of the electrode = 1 mm
- external radius of the nerve = 1.25 mm
- length of the electrode = 5 mm
- thickness of the electrode = 0.1 mm
- number of active site = 0 mm
- depth of active site = 0.05 mm
- diameter of active site = 0.25 mm
- z-position of active site = 0
- angle of insertion around z axis = 0

Point Sources

In the figure several grid of electrode of type POINT SOURCES is represented using COMSOL object

Parameters:

- n. of points = 16

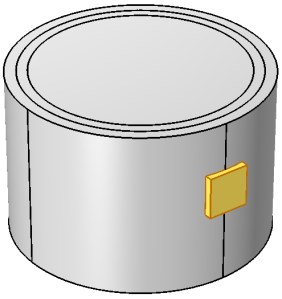


Transcutaneous Electrode

In the figure an electrode of type TRANSCUTANEOUS ELECTRODE is represented using COMSOL object

Parameters:

- width = 15
- depth = 6
- height = 15



Appendix B

Nerves

Nerves Types

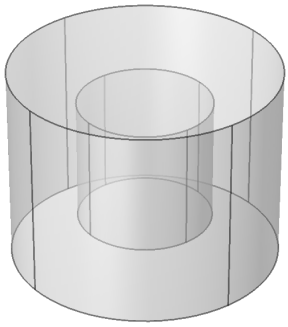
The HMLab allows you to choose among different types of nerves morphology:

Empty Nerve

In the figure an Empty Nerve, immersed in a Saline bath, is represented using COMSOL object

Parameters:

- Nerve extrusion length = 18.9603 mm
- Nerve radius = 10 mm
- Saline radius = 20 mm
- Saline vertical buffer = 30 mm



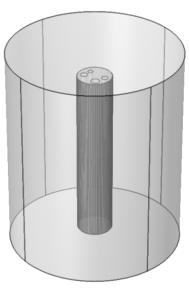
Peripheral Nerve with Circular Fascicles

In the figure an Peripheral Nerve with Circular Fascicles, immersed in a Saline bath,is represented using COMSOL object

Parameters:

- number of fascicles = 4
- Saline radius = 5 mm
- Saline vertical buffer = 12 mm

All the parameters used for generating the nerve structure and the fascicles structure are taken from thid MATLAB file: nerve_morphology.mat



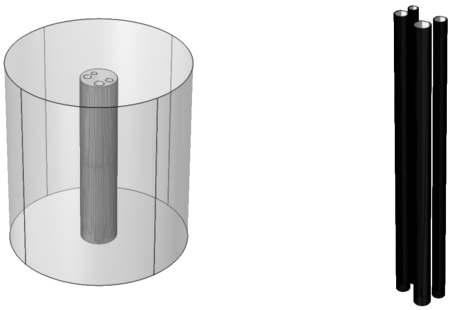
Peripheral Nerve with Polylinear Fascicles

In the figure an Peripheral Nerve with Polylinear Fascicles, immersed in a Saline bath,is represented using COMSOL object

Parameters:

- number of fascicles = 4
- Saline radius = 5 mm
- Saline vertical buffer = 12 mm

All the parameters used for generating the nerve structure and the fascicles structure are taken from thid MATLAB file: nerve_morphology_poly.mat



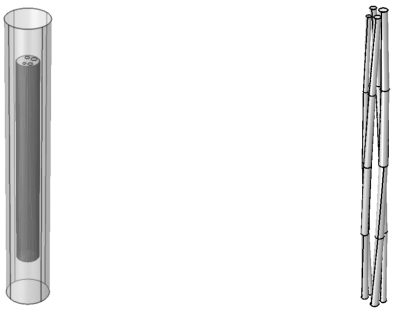
Peripheral Nerve with Rotating Fascicles

In the figure an Peripheral Nerve with Rotating Fascicles, immersed in a Saline bath,is represented using COMSOL object

Parameters:

- number of fascicles = 4
- angle of rot = 30°
- n. of section = 4
- Saline radius = 2 mm
- Saline vertical buffer = 28 mm

All the parameters used for generating the nerve structure and the fascicles structure are taken from thid MATLAB file: nerve_morphology.mat



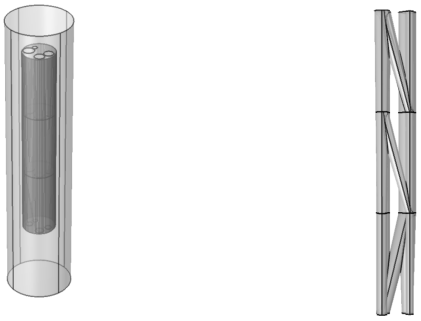
Peripheral Nerve with Merging and Splitting Fascicles

In the figure an Peripheral Nerve with Merging and Splitting Fascicles, immersed in a Saline bath,is represented using COMSOL object

Parameters:

- number of fascicles = 4
- section = [[0,[1,0],2,3],[0,1,[2,3],3],[0,1,[2,1,3],3]]
- n. of section = 3
- Saline radius = 2 mm
- Saline vertical buffer = 18 mm

All the parameters used for generating the nerve structure and the fascicles structure are taken from thid MATLAB file: nerve_morphology.mat



Appendix C

Simulations

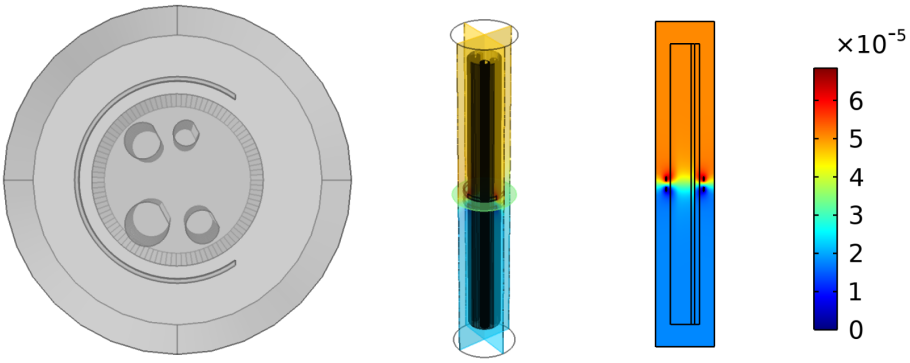
HMLab Example

Peripheral Nerve with Polylinear Fascicles and CortecCuff Electrode

You can run this file python/params.py :

```
from FEMModel import*
class Electrodes():
    def __init__(self):
        self.radius = -1 # radius of the electrode
        self.radius_nerve = -1 # external radius of the nerve
        self.length = -1# length of the electrode
        self.thick = -1 # thickness of the electrode
        self.length_as = -1 # length of active site
        self.depth_as = -1 # depth of active site
        self.width_as = -1 # width of active site
        self.ell_cc= -1 # center-to-center distance between sites
        self.z = -1 # z position of active sites
        self.theta_z = -1 # angle of insertion around z axis
        self.x_displace = 0
        self.d_as = -1
        self.h_as = -1
        self.n_as = -1
        self.d_rings = -1
        self.l_shaft = -1
        self.w_shaft = -1
        self.h_shaft = -1
        self.l_cc = -1
        self.x = -1
        self.y = -1
        self.theta_x = -1
        self.theta_y = -1


class Param():

    def __init__(self):
        self.n_elecs = [1]
        self.implant_type = 'Cortec Cuff'
        self.electrodes = [Electrodes()]
        self.radius = 1*1e-3
        self.radius_nerve = 1.25*1e-3
        self.electrodes[0].length = 5*1e-3
        self.electrodes[0].thick = 0.1*1e-3
        self.electrodes[0].length_as = 2.5*1e-3
        self.electrodes[0].depth_as =0.05*1e-3
        self.electrodes[0].width_as =0.25*1e-3
        self.electrodes[0].ell_cc = 0.7*1e-3
        self.electrodes[0].z = 0
        self.electrodes[0].theta_z =0
        self.electrodes[0].x_displace =0
        self.saline_type= 'Saline'
        self.nerve_type = 'peripheral'
        self.topo_type = 'polylinear'
        self.nerve_extrusion_length = 18 *1e-3#18.9603*1e-3
        self.nerve_filename = 'nerve_morphology_poly.mat'
        self.radius_ext = 2*1e-3
        self.topbot = 3*1e-3
        self.elec_filename = 'LOCS2.mat'

    def get_elec(self):
        for j in range(self.n_elecs[0]):
            self.electrodes.append(Electrodes())




 if __name__ == '__main__':

     params = Param()
     femmodel = FEMModel()
     femmodel.generate_materials()
     femmodel.get_custom_params(params)
     femmodel.generate_geometry()
     femmodel.assign_materials()
     femmodel.model.save('poly_cuff')
```

Here you can find the resulted model, using a 1e-6A current comsol/poly_cuff.mph :

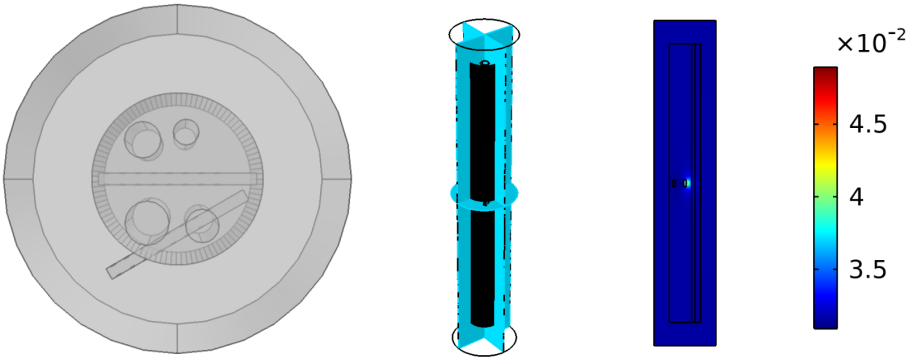


Peripheral Nerve with Circular Fascicles with 2 TIME Electrodes

You can change the main with different parameters :

```
if __name__ == '__main__':

    params = Param()
    params.implant_type = 'TIME'
    params.electrodes[0].l_shaft = 2*1e-3
    params.electrodes[0].w_shaft = 0.5  *1e-3
    params.electrodes[0].h_shaft = 0.15 *1e-3
    params.electrodes[0].l_cc = 0.2  *1e-3
    params.electrodes[0].n_as = 8
    params.electrodes[0].d_as = 0.07 *1e-3
    params.electrodes[0].h_as = 0.01 *1e-3
    params.electrodes[0].x = 0 *1e-3
    params.electrodes[0].y = 0 *1e-3
    params.electrodes[0].z = 0 *1e-3
    params.electrodes[0].theta_x = float(0)
    params.electrodes[0].theta_y = float(0)
    params.electrodes[0].theta_z = float(0)
    params.electrodes[1].l_shaft = 2*1e-3
    params.electrodes[1].w_shaft = 0.5  *1e-3
    params.electrodes[1].h_shaft = 0.15 *1e-3
    params.electrodes[1].l_cc = 0.2  *1e-3
    params.electrodes[1].n_as = 8
    params.electrodes[1].d_as = 0.07 *1e-3
    params.electrodes[1].h_as = 0.01 *1e-3
    params.electrodes[1].x = 0 *1e-3
    params.electrodes[1].y = -0.7*1e-3
    params.electrodes[1].z = 0 *1e-3
    params.electrodes[1].theta_x = float(0)
    params.electrodes[1].theta_y = float(0)
    params.electrodes[1].theta_z = float(30)
    params.topo_type = 'circular'
    params.nerve_filename = 'nerve_morphology.mat'
    femmodel = FEMModel()
    femmodel.generate_materials()
    femmodel.get_custom_params(params)
    femmodel.generate_geometry()
    femmodel.assign_materials()
    femmodel.model.save('TIME__2')
```

Here you can find the resulted model, using a 1e-6A current comsol/TIME__2.mph :

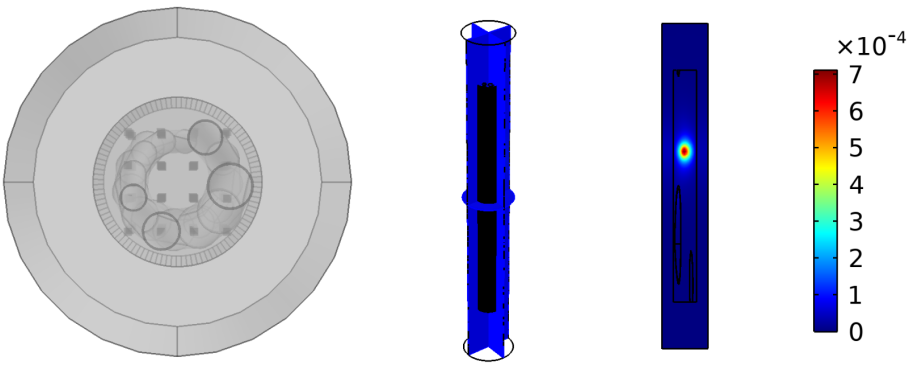


Peripheral Nerve with Rotating Fascicles with 4 Grid Point Sources

You can change the main with different parameters :

```
if __name__ == '__main__':

     params = Param()
     params.n_elecs = [4]
     params.implant_type = 'Point sources'
     params.electrodes[0].elec_filename = 'LOCS3.mat'
     params.nerve_type = 'rotating fascicles'
     params.nerve_filename = 'nerve_morphology.mat'

     femmodel = FEMModel()
     femmodel.generate_materials()
     femmodel.get_custom_params(params)
     femmodel.generate_geometry()
     femmodel.assign_materials()
     femmodel.model.save('fasc_rot')
```

Here you can find the resulted model, using a 1e-6A current comsol/fasc_rot.mph :

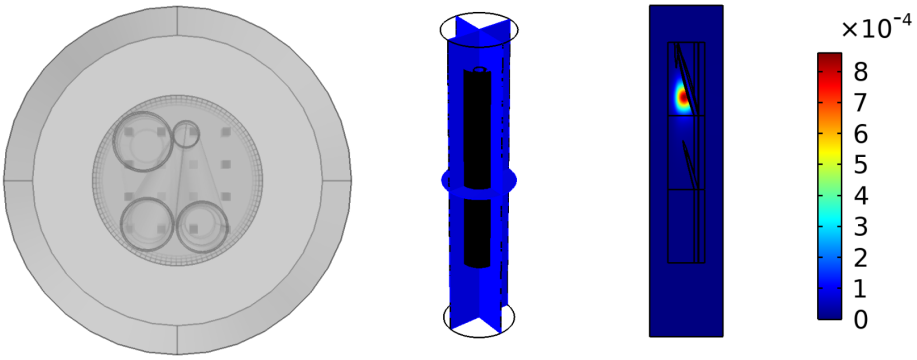


Peripheral Nerve with Split-Merge Fascicles with 4 Grid Point Sources

You can change the main with different parameters :

```
if __name__ == '__main__':

     params = Param()
     params.n_elecs = [4]
     params.implant_type = 'Point sources'
     params.electrodes[0].elec_filename = 'LOCS3.mat'
     params.nerve_type = 'Splitting fascicles'
     params.nerve_filename = 'nerve_morphology.mat'

     femmodel = FEMModel()
     femmodel.generate_materials()
     femmodel.get_custom_params(params)
     femmodel.generate_geometry()
     femmodel.assign_materials()
     femmodel.model.save('split_fasc')
```

Here you can find the resulted model, using a 1e-6A current comsol/split_fasc.mph :



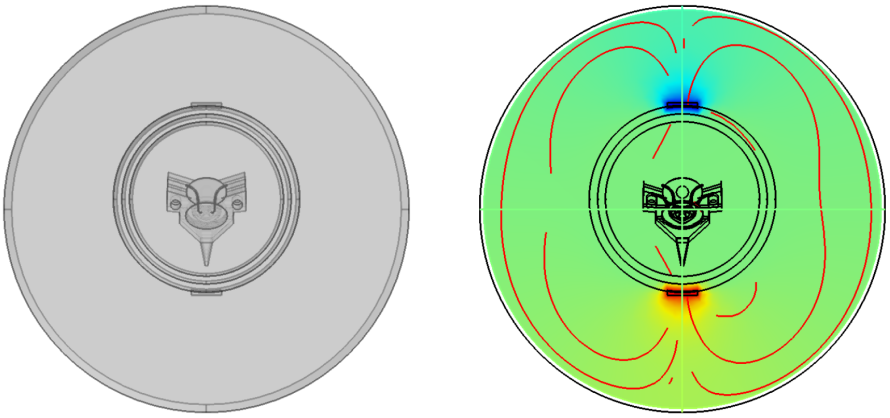
To obtain the 2D plot look at Comsol documentation
https://cdn.comsol.com/doc/6.1.0.346/COMSOL_ProgrammingReferenceManual.pdf

Transcutaneous Stimulation of Spinal Cord

You can run this file python/ex_trans.py :

```
from FEM import FEModel

if __name__ == '__main__':
    fem_model = FEModel()
    identifier = 1
    ##################
    cross_section_names = ['C5','C6','C7','C8'];
    size_motoneurons = [9, 4];
    size_groupIaneurons = [1,3];
    material_names= ["gm","wm","CSF","dura","fat"]
    muscle_names=['']
    ###################

    fem_model.spinal_cord.set_params(cross_section_names, size_motoneurons,
                                     size_groupIaneurons, muscle_names, material_names)
    offset_up_and_down = .1*fem_model.spinal_cord.h_sal
    start=fem_model.spinal_cord.min_heigth_absolute-offset_up_and_down
    radius=fem_model.spinal_cord.max_radius_absolute*2.1
    fem_model.air.set_params(100,60)
    fem_model.body.set_params(radius*1.1, radius, (((radius*1.1)*1.1)*1.1)*1.1,
                              (radius*1.1)*1.1, start, ((radius*1.1)*1.1)*1.1,60)
    fem_model.implant.set_params_default()
    fem_model.generate_geometry()
    fem_model.add_physics_default()
    fem_model.assign_materials()
```

Here you can find the resulted model, using a 1A current comsol/trans.mph :



Bibliography

[1] Comsol multiphysics, . URL https://www.comsol.it/comsol-multiphysics.

[2] Comsol programming reference manual, .

[3] Mph read the docs. URL https://mph.readthedocs.io/en/stable/api/mph.Model.html#mph.Model.evaluate.

[4] Pytorch data. URL https://pytorch.org/docs/stable/data.html.

[5] Sphinx documentation. URL https://www.sphinx-doc.org/en/master/index.html.

[6] Xgboost python api. URL https://xgboost.readthedocs.io/en/latest/python/python_api.html.

[7] scikit-image. URL https://scikit-image.org/.

[8] Comsol java api reference guide, 2013. URL http://lmn.pub.ro/~daniel/ElectromagneticModelingDoctoral/Books/COMSOL4.3/mph/COMSOLJavaAPIReferenceGuide.pdf. Accessed on October 10, 2023.

[9] *Frontiers in Neurology*, 2018.

[10] V. Anusuya, J. Sharan, and AK. Jena. Morphometric characteristics of cervical vertebrae in subjects with short, normal, and long faces. *Surgical and Radiologic Anatomy*, 43:865–872, 2021.

[11] O. Bican, A. Minagar, and AA. Pruitt. The spinal cord: a review of functional neuroanatomy. *Neurologic Clinics*, 31:1–18, 2013. doi: 10.1016/j.ncl.2012.09.009.

[12] M. Capogrosso, A. Autore1, B. Autore2, and C. Autore3. A computational model for epidural electrical stimulation of spinal sensorimotor circuits. *Journal of Neuroscience*, 33:19326–19340, 2013. doi: 10.12345/js.2013.1.

[13] B. De Leener, VS. Fonov, D. Louis Collins, V. Callot, N. Stikov, and J. Cohen-Adad. PAM50: Unbiased multimodal template of the brainstem and spinal cord aligned with the ICBM152 space. *Neuroimage*, 2018.

[14] K. Garcia, J.K. Wray, and S. Kumar. Spinal cord stimulation, 2023. URL https://www.ncbi.nlm.nih.gov/books/NBK553154/. Updated on April 24, 2023.

[15] A. Hirata, Y. Takano, Y. Kamimura, and O. Fujiwara. Effect of the averaging volume and algorithm on the in situ electric field for uniform electric- and magnetic-field exposures. *Physics in Medicine & Biology*, 55:N243–N252, 2010.

[16] J. Holsheimer, JA. den Boer, JJ. Struijk, and AR. Rozeboom. Mr assessment of the normal position of the spinal cord in the spinal canal. *AJNR. American journal of neuroradiology*, 15:951–959, 1994.

[17] AR. Jackson, F. Travascio, and WY. Gu. Effect of mechanical loading on electrical conductivity in human intervertebral disc. *Journal of Biomechanical Engineering*, 131:054505, 2009.

[18] J. Ladenbauer, K. Minassian, US. Hofstoetter, MR. Dimitrijevic, and F. Rattay. Stimulation of the human lumbar spinal cord with implanted and surface electrodes: a computer simulation study. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 18:637–645, 2010. doi: 10.1109/TNSRE.2010.2054112.

[19] L. Liang, A. Damiani, M. Del Brocco, E.R. Rogers, M.K. Jantz, L.E. Fisher, R.A. Gaunt, M. Capogrosso, S.F. Lempka, and E. Pirondini. A systematic review of computational models for the design of spinal cord stimulation therapies: from neural circuits to patient-specific simulations. *Journal of Physiology*, 601:3103–3121, 2023. doi: 10.1113/JP282884.

[20] MD Michael W. Devereaux. *Anatomy and Examination of the Spine*. Medical Publishing Company, Cleveland, OH, 2023. ISBN: 123-456-7890.

[21] A. Nunès, G. Glaudot, A. Lété, A. Balci, B. Lengelé, C. Behets, and A. Jankovski. Measurements and morphometric landmarks of the human spinal cord: A cadaveric study. *Clinical Anatomy*, 36:631–640, 2023. doi: 10.1002/ca.24010.

[22] Seong-Hoon Oh, Noel I. Perin, and Paul R. Cooper. Quantitative three-dimensional anatomy of the subaxial cervical spine. implication for anterior spinal surgery. *Journal of Biomechanical Engineering*, 139:0645011–0645017, 2000.

[23] G. Prabavathy, Chandra Philip X., G. Arthi, and T. Sadeesh. Morphometric study of cervical vertebrae c3-c7 in south indian population – a clinico-anatomical approach. *Italian Journal of Anatomy and Embryology*, 122:49–57, 2017.

[24] D. Purves, GJ. Augustine, D. Fitzpatrick, et al., editors. Sinauer Associates, 2001.

[25] Rakesh Ranjan, Md. Zahid Hussain, Soni Kumari, Vijay Kumar Singh, and Rashmi Prasad. The morphology and incidence of the accessory foramen transversarium in human dried cervical vertebrae as well as their clinical significance in the eastern indian population. *Asian Journal of Medical Sciences*, 13:47–53, 2022.

[26] S. Romeni, G. Valle, A. Mazzoni, et al. A computational framework for the design and optimization of peripheral neural interfaces. *Nature Protocols*, 15:3129–3153, 2020. doi: 10.1038/s41596-020-0377-6.

[27] Olaf Ronnenberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation.

[28] T. Shimizu, S. Pongmanee, and KD. Riew. Inter-spinous process distance: a novel parameter predicting segmental lordosis during posterior cervical spine deformity surgery. *European Spine Journal*, 28:1192–1199, 2019.

[29] R. Veeramani, K. Thangarasu, and U. Amirthalingam. Morphometry of the uncinate process, vertebral body, and lamina of the c3–7 vertebrae relevant to cervical spine surgery. *Neurospine*, 2019.

[30] Y. Yu, H. Mao, J-S. Li, T-Y. Tsai, L. Cheng, KB. Wood, G. Li, and TD. Cha. Ranges of cervical intervertebral disc deformation during an in vivo dynamic flexion–extension of the neck. *Journal of Biomechanical Engineering*, 139: 0645011–0645017, 2017.