ASCENT

Release v1.1.3

Musselman ED, Cariello JE, Grill WM, Pelot NA.

Jul 26, 2022

BASIC ASCENT USAGE

Please check out the associated publication in PLOS Computational Biology!

Cite both the paper and the DOI for the release of the repository used for your work. We encourage you to clone the most recent commit of the repository.

- **Cite the paper:**

  APA

  Musselman, E. D., **Cariello, J. E.**, Grill, W. M., & Pelot, N. A. (2021). ASCENT (Automated Simulations to Characterize Electrical Nerve Thresholds): A pipeline for sample-specific computational modeling of electrical stimulation of peripheral nerves. PLOS Computational Biology, 17(9), e1009285. https://doi.org/10.1371/journal.pcbi.1009285

  MLA

  Musselman, Eric D., et al. "ASCENT (Automated Simulations to Characterize Electrical Nerve Thresholds): A Pipeline for Sample-Specific Computational Modeling of Electrical Stimulation of Peripheral Nerves." PLOS Computational Biology, vol. 17, no. 9, Sept. 2021, p. e1009285. PLoS Journals, https://doi.org/10.1371/journal.pcbi.1009285.

  BibTeX

```
@article{Musselman2021,
  doi = {10.1371/journal.pcbi.1009285},
  url = {https://doi.org/10.1371/journal.pcbi.1009285},
  year = {2021},
  month = sep,
  publisher = {Public Library of Science ({PLoS})},
  volume = {17},
  number = {9},
  pages = {e1009285},
  author = {Eric D. Musselman and Jake E. Cariello and Warren M. Grill and Nicole A.
↪ Pelot},
  editor = {Dina Schneidman-Duhovny},
  title = {{ASCENT} (Automated Simulations to Characterize Electrical Nerve␣
↪Thresholds): A pipeline for sample-specific computational modeling of electrical␣
↪stimulation of peripheral nerves},
  journal = {{PLOS} Computational Biology}
}
```

- **Cite the code (use the DOI for the version of code used):**

  APA

  Musselman, E. D., **Cariello, J. E.**, Grill, W. M., & Pelot, N. A. (2022). wmglab-duke/ascent: ASCENT v1.1.3 (v1.1.3) [Computer software]. Zenodo. https://doi.org/10.5281/ZENODO.6537277

  MLA

  Musselman, Eric D., et al. Wmglab-Duke/Ascent: ASCENT v1.1.3. v1.1.3, Zenodo, 2022, doi:10.5281/ZENODO.6537277.

  BibTeX

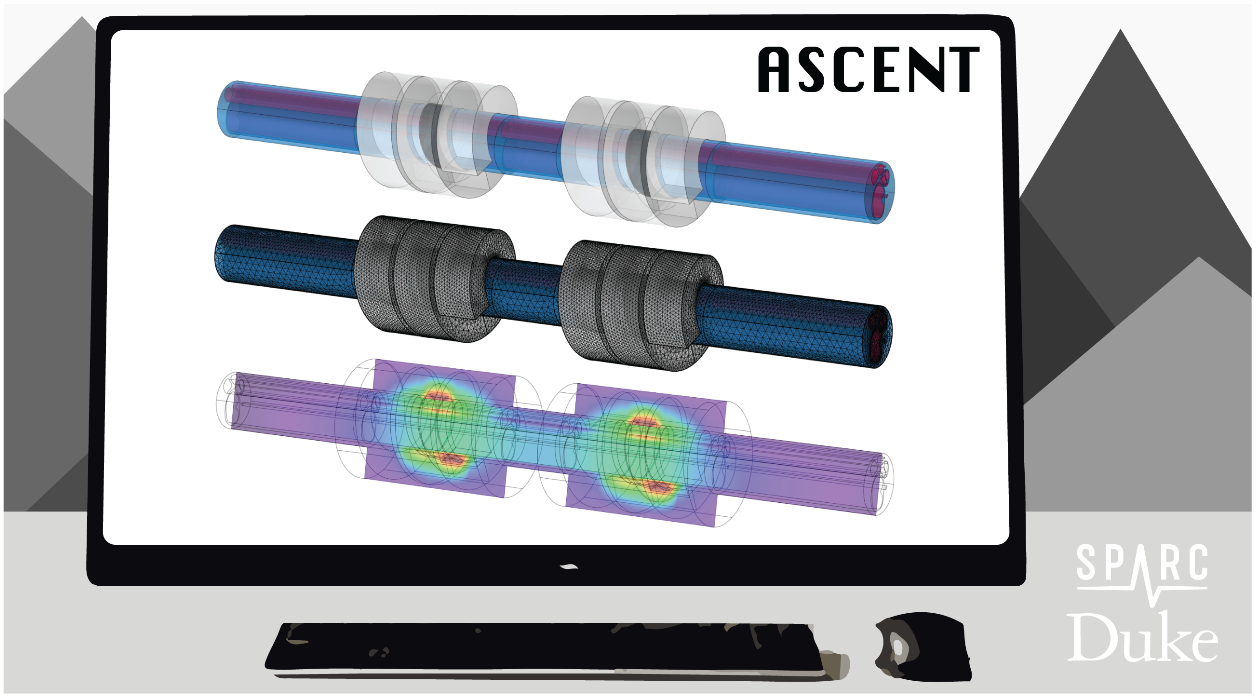
```
@misc{https://doi.org/10.5281/zenodo.6537277,
  doi = {10.5281/ZENODO.6537277},
  url = {https://zenodo.org/record/6537277},
  author = {Musselman,  Eric D and Cariello,  Jake E and Grill,  Warren M and Pelot,
↪  Nicole A},
```

(continues on next page)

(continued from previous page)

```
  title = {wmglab-duke/ascent: ASCENT v1.1.3},
  publisher = {Zenodo},
  year = {2022},
  copyright = {MIT License}
}
```

ASCENT is an open source platform for simulating peripheral nerve stimulation. To download the software, visit the ASCENT GitHub repository.



CHAPTER

ONE

GETTING STARTED

1.1 Installation

1.1.1 Installing commercial software

It is *highly* recommended that you use a distribution of Anaconda/Miniconda with ASCENT. However, advanced users who wish to use another Python distribution may.

First, these software packages must be manually installed:

- Miniconda/Anaconda We recommend that you install Miniconda (Miniconda is a stripped down version of Anaconda; Anaconda is optional for intermediate users). If you alread have an existing installation, there is no need to reinstall.
  - Recommended: Select add to path
  - Recommended: Select "Install for individual user"
  - https://docs.conda.io/en/latest/miniconda.html
  - https://www.anaconda.com/products/individual
- Java SE Development Kit 8 (1.8) (need to register for a free account)
  - https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html
- Homebrew (Mac and Linux only)
  - `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
  - `brew install wget`
- COMSOL 5.4 or newer (requires purchase of license; only based package needed, which includes the COMSOL Java API)
  - Once COMSOL is installed, alter 'File System Access' permissions via File → Preferences → Security → Methods and Java Libraries → File System Access → All Files.
  - Open the COMSOL Server and log in with a username and password of your choosing (arbitrary and not needed thereafter). This can be done by navigating to the bin/ directory in the COMSOL installation and running `comsolmphserver` (Windows) or `./comsol server` (MacOS/Linux).
  - https://www.comsol.com/product-download
- NEURON 7.6 (newer versions have been released, but compatibility has yet to be confirmed; choose appropriate installer depending on operating system; install auxiliary software as prompted by NEURON installer)
  - https://neuron.yale.edu/ftp/neuron/versions/v7.6/

    - If having issues with the NEURON installation, try running the compatibility troubleshooter.

In this stage of development, all programs/commands are run from a command line environment on both MacOS/Linux and Windows operating systems (Bash terminal for MacOS/Linux, Powershell for Windows). For users less familiar with this environment and for the quickest setup, it is suggested that the user install the package management system Miniconda or Anaconda (if using MacOS, choose .pkg for ease of use).

Note that compiling NEURON files (when submitting sims) requires that the following packages be installed and on your path (Mac and Linux ONLY):

- openmpi-2.0.0
- libreadlines.so.6

If using MacOS to run local NEURON simulations, it may be necessary to install the Xcode Command Line Tools via `xcode-select --install`, as well as Xquartz.

Users may also download a text editor or integrated development environment (IDE) of their choosing to view/edit code (e.g., Atom, Visual Studio Code, IntelliJ IDEA). For Java code, full autocomplete functionality requires adding both the path to the COMSOL installation ending in `plugins` as well as the path `<ASCENT_PATH>/bin/json-20190722.jar` to the list of available libraries (usually from within the IDE's project settings).

1.1.2 Installing ASCENT

1. First, download or clone the SPARC ASCENT pipeline from GitHub to a desired location that will be referenced in step 3. Downloading is a much simpler process than cloning via Git, but does not easily allow for you to get the most recent updates/bug fixes, nor does it allow you to suggest new features/changes. If you are interested in either of these features, you should clone via Git rather than downloading.

    - Downloading: Click the download button on GitHub and choose the location to which you would like to save. Note that you will need to extract the files, as they will be downloaded in a compressed format. When presented with a choice of compression format, ".zip" is a safe choice that most computers will be able to extract.
    - Cloning via Git:
        1. You must first have an account with GitHub, and if you have been granted special permissions to the code repository, you must use the email address to which those permissions were granted.
        2. If you have not already done so, add an SSH key to your account (see instructions for GitHub). This is a required standard authentication method.
        3. In a Miniconda (or Git, for advanced users) command line environment, navigate to the location to where you would like to clone the code repository (see instructions for navigating the file system from the command line for Mac or Linux and Windows).
        4. Clone the repository (see instructions for GitHub).
        5. For more information on using Git, check out the official documentation.

2. Next, install ASCENT dependencies:

    - Windows: Open the Anaconda Powershell Prompt from the Windows Start Menu as Administrator, and use cd to navigate to the root directory of the pipeline. Then, run `python run install`. Note: If reinstalling ASCENT after having changed your Anaconda/Miniconda installation, it may be necessary to delete the `#region conda initialize` block from `C:\Users\<username>\Documents\WindowsPowerShell\profile.ps1` in order to use Anaconda Powershell Prompt.
    - MacOS/Linux: Open Anaconda Prompt and use cd to navigate to the root directory of the pipeline. Then, run `python run install`.
    - For advanced users using their own (non-conda) Python distribution:

  - From the ascent root directory execute `python run install --no-conda`
  - From the ascent root directory execute `pip install -r requirements.txt`
  - This method is highly discouraged as newer versions of packages/Python could potentially break ASCENT or introduce unexpected bugs

After confirming that you are in the correct directory, the script will install the required Python packages: Pillow [Clark, 2015], NumPy [Harris *et al.*, 2020], Shapely [Gillies and others, 2007–], Matplotlib [Hunter, 2007], PyClipper [Chalton and others, 2015–], pygame [Shinners, 2011–], QuantiPhy [Kundert, 2016–], OpenCV [Bradski, 2000], PyMunk [Blomqvist, 2007], and SciPy [Virtanen *et al.*, 2020], pandas [pandas development team, 2020, Wes McKinney, 2010], OpenPyXL [Gazoni and Clark, 2020–], scikit-image [Van der Walt *et al.*, 2014], ffmpeg [Tomar, 2006], xlsxwriter [McNamara, 2017–], seaborn [Waskom, 2021]. It is crucial that all of the packages are installed successfully (check for "successfully installed" for each package). The installation script will also create a shortcut to the newly configured ASCENT Conda "environment" on your project path, which can be used for running the pipeline.

1. Then, configure the environment variables. This step may be completed several ways, described below.
   - Recommended Setup: Open Anaconda prompt, navigate to the ASCENT root directory, and execute `python run env_setup`. You will be prompted for the following paths:
     - ASCENT_COMSOL_PATH: Path to the COMSOL installation, ending in the directory name "Multiphysics", as seen in the template and *JSON Overview*.
     - ASCENT_JDK_PATH: Path to the JDK 1.8 installation, ending in `bin`, as seen in the template and *JSON Overview*. Hint: This is the correct path if the directory contains many executables (for Windows: java.exe, etc.; MacOS/Linux: java, etc.).
     - ASCENT_PROJECT_PATH: Path to the root directory of the pipeline, as chosen for step 1.
     - ASCENT_NSIM_EXPORT_PATH: Path to the export location for NEURON "simulation" directories. This path only depends on on the user's desired file system organization.
   - Manual Setup: Copy the file `config/templates/env.json` into `config/system/env.json` (new file). This file holds important paths for software used by the pipeline (see env.json in *Enums* and *JSON Overview*). Then, edit each of the four values as specified below. Use \\ in Windows and / in MacOS/Linux operating systems. Note that the file separators are operating system dependent, so even if you installed in step 2 with Unix-like command environment on a Windows machine (e.g., using Git Bash, Cygwin, or a VM with Ubuntu), you will still need to choose the proper file separator for Windows, i.e., \\). See example env.json files for both MacOS and Windows (*Environment Parameters*).
   - Automatic setup: Upon the initiation of your first run, you will be prompted to enter the above four paths if you did not choose to complete the manual setup. Enter them as prompted, following the guidelines detailed above and exemplified in *JSON Overview*. Note that you may at any time update paths with `python run env_setup` to rewrite this file if the information should change.

1.2 Metadata required to model an in vivo experiment using the ASCENT pipeline

Note: All metadata required for the *tutorial run* are provided with ASCENT.

1. Detailed specifications / dimensions of the stimulating cuff electrode.
2. Transverse cross section of the nerve where the cuff is placed, stained to visualize the different tissue types (e.g., using Masson's trichrome), with a scale bar (Fig 2 and Running_ASCENT/Info.md#morphology-Input-Files)) or known scale (micrometers/pixel). Different possible sources for defining the nerve sample morphology include:

   a. For best specificity, the nerve would be sampled from the specific animal used in the experiment being modeled. In this case, two colors of tissue dye may be used on the ventral and medial aspects of the nerve to maintain orientation information.

   b. Otherwise, a sample from another animal of the same species could be used at the correct nerve level.

   c. If multiple samples from other animals are available, they could be used to generate a representative nerve model, knowing the range of morphological metrics across individuals using the `scripts/mock_morphology_generator.py` script (*Mock Morphology*).

   d. Lastly, published data could be used.

3. Orientation and rotation of the cuff on the nerve (e.g., cuff closure on the ventral side of the nerve).
4. Fiber diameters

   a. Distributions of fiber diameters may be obtained from literature; otherwise, detailed electromicroscopic studies are required.

   b. The fiber diameters found in the target nerve that will be simulated in NEURON. All diameters or a subset of diameters may be of interest.

   c. Each fiber diameter of interest can be simulated for each fiber location of interest, or specific fiber diameters can be simulated in specific locations.

5. Approximate tissue or fluids surrounding the nerve and cuff (e.g., muscle, fat, or saline).
6. Stimulation waveforms, pulse widths, and other parameters of the electrical signal.
7. If comparing to neural recordings: distance between the stimulation and recording cuffs.
8. If comparing to functional recordings (e.g., EMG): distance from the stimulation cuff to the location where the nerve inserts into the muscle.

1.3 Tutorial Run

Following the instructions below task and verify the threshold value to familiarize yourself with the ASCENT code and documentation.

We provide segmented histology of a rat cervical vagus nerve (`examples/tutorial/`). Use the provided histology and configurations files to simulate activation thresholds in response to a charge balanced, biphasic pulse (PW1 = 100 s, interphase gap of 100 s, PW2 = 400 s) using Purdue's bipolar cuff design.

- MRG 8.7 m diameter fibers
- Fibers placed in nerve cross section using a 6 spoke wheel with 2 fibers per spoke
- Custom material for surrounding medium with isotropic conductivity 1/20 [S/m]

After your thresholds have been computed, build a heatmap for the threshold at each fiber location using the example script: `examples/analysis/heatmap.py`.

Through this exercise, you will:

- Place and name binary masks of the nerve morphology in the proper directories
  - Binary masks provided
- Define and assign a custom material
- Build and solve a finite element model
- Define placement of fibers in the nerve cross section

- Parameterize your custom stimulation waveform
- Simulate activation thresholds for a specific fiber model by submitting NEURON simulations locally or to a computer cluster
- Generate a heatmap of fiber activation thresholds

Check: Threshold for inner0_fiber0 (`thresh_inner0_fiber0.dat`) should be -0.027402 mA

We provided ***Sample***, ***Model***, and ***Sim*** JSON files for the solution in `examples/tutorial/`. Use the steps below in order to set up this tutorial run.

1.4 Setting up a run of ASCENT

How to run the ASCENT pipeline, after completing the initial setup.

To set up a run of the pipeline, you must provides binary mask inputs for the nerve and save ***Sample*** (i.e., `sample.json`), ***Model(s)*** (i.e., `model.json`), and ***Sim(s)*** (i.e., `sim.json`) JSON configurations in directories, relative to the project path defined in `config/system/env.json`. The directory names must use indices that are consistent with the indices of ***Sample***, ***Model(s)***, and ***Sim(s)*** defined in ***Run***.

1. ***Masks:*** Populate `input/<NAME>/` (e.g., "Rat1-1", which must match "sample" parameter in ***Sample***) with binary masks of neural tissue boundaries using either:

   a. Segmented histology (Running_ASCENT/Info.md#morphology-Input-Files) and Fig 2), or

   b. The `mock_morphology_generator.py` script (*Mock Morphology*).

      1. Copy `mock_sample.json` from `config/templates/` to `config/user/mock_samples/` as `<mock_sample_index>.json` and update file contents, including the "NAME" parameter used to construct the destination path for the output binary masks, which serve as inputs to the pipeline.
      2. Call `"python run mock_morphology_generator <mock_sample_index>"`.
      3. The program saves a copy of your `mock_sample.json` and binary masks in `input/<NAME>/`.

2. ***For one Sample:*** Copy `sample.json` from `config/templates/` to `samples/<sample_index>/` as `sample.json` and edit its contents to define the processing of binary masks to generate the two-dimensional cross section geometry of the nerve in the FEM. In particular, change "sample" to match `<NAME>`, the `"scale_bar_length"` parameter for `s.tif` (i.e., length in microns of your scale bar, which is oriented horizontally), and `"mask_input"` in ***Sample*** accordingly (*Sample Parameters*). You have now created the directory for your first sample: `sample #<sample_index>`. Note: in lieu of a scale bar image, the user may optionally specify the microns/pixel ratio for the sample mask(s).

3. ***For each Model:*** Copy `model.json` from `config/templates/` to `samples/<sample_index>/models/<model_index>/` as `model.json` and edits its contents to define the three dimensional FEM.

   a. Assign the cuff geometry to be used by placing a file name from `config/system/cuffs` in `"cuff":"preset"`.

   b. Optionally, define a new cuff geometry specific to your needs:

      1. ***Preset:*** User defines a new "preset" cuff JSON file, which contains instructions for creating their cuff electrode, and saves it as `config/system/cuffs/<preset_str>.json`.
      2. The `<preset_str>.json` file name must be assigned to the "preset" parameter in ***Model*** (*Model Parameters*).

4. ***For each Sim:*** Copy `sim.json` from `config/templates/` to `config/user/sims/` as `<sim_index>.json` and edit its contents to inform the NEURON simulations (*Sim Parameters*).

5. ***Run:*** Copy `run.json` from `config/templates/` to `config/user/runs/` as `<run_index>.json` and edit the indices for the created ***Sample***, ***Model(s)***, and ***Sim(s)*** configurations (*Run Parameters*).

6. The pipeline is run from the project path (i.e., the path to the root of the ASCENT pipeline, which is defined in `config/system/env.json`) with the command `"python run pipeline <run indices>"`, where `<run indices>` is a list of space-separated ***Run*** indices (if multiple ***Sample*** indices, one ***Run*** for each). The pipeline outputs ready-to-submit NEURON simulations and associated ***Run file(s)*** to the `"ASCENT_NSIM_EXPORT_PATH"` directory as defined in `config/system/env.json` (*Environment Parameters*). NEURON simulations are run locally or submitted to a computer cluster with the command `"python submit.py <run indices>"` from the export directory.

CHAPTER

TWO

RUNNING ASCENT

2.1 How to use ASCENT

2.1.1 Running the ASCENT Pipeline

See *Running the Pipeline*.

2.1.2 Submitting NEURON jobs

We provide scripts for the user to submit NEURON jobs in `src/neuron/`. The `submit.py` script takes the input of the ***Run*** configuration and submits a NEURON call for each independent fiber within an `n_sim/`. These scripts are called using similar syntax as `pipeline.py`: `"./submit.<ext> <run indices>,"` where `<run indices>` is a space-separated list of integers. Note that these submission scripts expect to be called from a directory with the structure generated by `Simulation.export_nsims()` at the location defined by `"ASCENT_NSIM_EXPORT_PATH"` in `env.json`.

Cluster submissions

When using a high-performance computing cluster running SLURM:

1. Set your default parameters, particularly your partition (can be found in `"config/system/slurm_params.json"`). The partition you set here (default is “common”) will apply to all `submit.py` runs.
2. Optionally, set your `"submission_context"` to `"auto"`, and configure your `run.json` with the appropriate hostname prefix. (Only necessary if you plan to submit runs both locally and on a cluster, see *Run Parameters*.)
3. Many parameters which `submit.py` sources from JSON configuration files can be overridden with command line arguments. For more information, see *Command-Line Arguments*.
4.

2.1.3 Other Scripts

We provide scripts to help users efficiently manage data created by ASCENT. Run all of these scripts from the directory to which you installed ASCENT (i.e., `"ASCENT_PROJECT_PATH"` as defined in `config/system/env.json`). For specific controls associated with each script, see *Command Line Arguments*.

`scripts/import_n_sims.py`

To import NEURON simulation outputs (e.g., thresholds, recordings of Vm) from your `ASCENT_NSIM_EXPORT_PATH` (i.e., `<"ASCENT_NSIM_EXPORT_PATH" as defined in env.json>/n_sims/<concatenated indices as sample_model_sim_nsim>/data/outputs/`) into the native file structure (see *ASCENT Data Hierarchy*, `samples/<sample_index>/models/<model_index>/sims/<sim_index>/n_sims/<n_sim_index>/data/outputs/`) run this script from your `"ASCENT_PROJECT_PATH"`.

```
python run import_n_sims <list of run indices>
```

The script will load each listed Run configuration file from `config/user/runs/<run_index>.json` to determine for which Sample, Model(s), and Sim(s) to import the NEURON simulation data. This script will check for any missing thresholds and skip that import if any are found. Override this by passing the flag `--force`. To delete n_sim folders from your output directory after importing, pass the flag `--delete-nsims`. For more information, see *Command-Line Arguments*.

`scripts/clean_samples.py`

If you would like to remove all contents for a single sample (i.e., `samples/<sample_index>/`) EXCEPT a list of files (e.g., `sample.json`, `model.json`) or EXCEPT a certain file format (e.g., all files ending `.mph`), use this script. Run this script from your `"ASCENT_PROJECT_PATH"`. Files to keep are specified within the python script.

```
python run clean_samples <list of sample indices>
```

`scripts/tidy_samples.py`

If you would like to remove ONLY CERTAIN FILES for a single sample (i.e., `samples/<sample_index>/`), use this script. This script is useful for removing logs, runtimes, special, and *.bat or *.sh scripts. Run this script from your `"ASCENT_PROJECT_PATH"`. Files to remove are specified within the python script.

```
python run tidy_samples <list of sample indices>
```

2.1.4 Data analysis tools

Python Query class

The general usage of Query is as follows:

1. In the context of a Python script, the user specifies the search criteria (think of these as "keywords" that filter your data) in the form of a JSON configuration file (see `query_criteria.json` in *Query Parameters*).
2. These search criteria are used to construct a Query object, and the search for matching ***Sample***, ***Model***, and ***Sim*** configurations is performed using the method `run()`.
3. The search results are in the form of a hierarchy of ***Sample***, ***Model***, and ***Sim*** indices, which can be accessed using the `summary()` method.

Using this "summary" of results, the user is then able to use various convenience methods provided by the Query class to build paths to arbitrary points in the data file structure as well as load saved Python objects (e.g., Sample and Simulation class instances).

The Query class's initializer takes one argument: a dictionary in the appropriate structure for query criteria (*Query Parameters*) *or* a string value containing the path (relative to the pipeline repository root) to the desired JSON configuration with the criteria. Put concisely, a user may filter results either manually by using known indices or automatically by using parameters as they would be found in the main configuration files. It is ***extremely important*** to note that the Query class must be initialized with the working directory set to the root of the pipeline repository (i.e., `sys.path.append(ASCENT_PROJECT_PATH)` in your script). Failure to set the working directory correctly will break the initialization step of the Query class.

After initialization, the search can be performed by calling Query's `run()` method. This method recursively dives into the data file structure of the pipeline searching for configurations (i.e., ***Sample***, ***Model***, and/or ***Sim***) that satisfy `query_criteria.json`. Once `run()` has been called, the results can be fetched using the `summary()` accessor method. In addition, the user may pass in a file path to `excel_output()` to generate an Excel sheet summarizing the Query results.

Query also has methods for accessing configurations and Python objects within the `samples/` directory based on a list of ***Sample***, ***Model***, or ***Sim*** indices. The `build_path()` method returns the path of the configuration or object for the provided indices. Similarly, the `get_config()` and `get_object()` methods return the configuration dictionary or saved Python object (using the Pickle package), respectively, for a list of configuration indices. These tools allow for convenient looping through the data associated with search criteria.

In addition, we have included a few data analysis methods in the Query class: `heatmaps()`, `barcharts_compare_models()`, and `barcharts_compare_samples()`. Since individual use cases for data analysis can differ greatly, these methods are not considered "core" functionality and can instead be treated as examples for how one might use the Query class. Example uses of these Query convenience methods are included in `examples/analysis/`.

- `plot_sample.py`
- `plot_fiberset.py`
- `plot_waveform.py`

Video generation for NEURON state variables

In `examples/analysis/` we provide a script, `plot_video.py`, that creates an animation of saved state variables as a function of space and time (e.g., transmembrane potentials, MRG gating parameters). The user can plot `n_sim` data saved in a `data/output/` folder by referencing indices for ***Sample***, ***Model***, ***Sim***, inner, fiber, and `n\_sim`. The user may save the animation as either a `*.mp4` or `*.gif` file to the `data/output/` folder.

The `plot_video.py` script is useful for determining necessary simulation durations (e.g., the time required for an action potential to propagate the length of the fiber) to avoid running unnecessarily long simulations. Furthermore, the script is useful for observing onset response to kilohertz frequency block, which is important for determining the appropriate duration of time to allow for the fiber onset response to complete.

Users need to determine an appropriate number of points along the fiber to record state variables. Users have the option to either record state variables at all Nodes of Ranvier (myelinated fibers) or sections (unmyelinated fibers), or at discrete locations along the length of the fiber (*Sim Parameters*).

2.2 Command-Line Arguments

Note: In cases where a behavior can be controlled by both a command line argument, and a configuration file (.json file), the command line argument will ALWAYS take precedence. run —

ASCENT: Automated Simulations to Characterize Electrical Nerve Thresholds

```
usage: run [-h] [-v] [-V] [-l {runs,samples,sims}]
           {pipeline,install,env_setup,clean_samples,import_n_sims,mock_morphology_
↪generator,tidy_samples}
           ...
```

2.2.1 Positional Arguments

script	Possible choices: pipeline, install, env_setup, clean_samples, import_n_sims, mock_morphology_generator, tidy_samples which script to run

2.2.2 Named Arguments

-v, --verbose	verbose printing Default: False
-V, --version	print version
-l, --list	Possible choices: runs, samples, sims List all available indices for the specified option

2.2.3 Sub-commands:

pipeline

main ASCENT pipeline

```
run pipeline [-h]
             [-b {pre_geom_run,post_geom_run,pre_java,post_mesh_distal,pre_mesh_distal,
↪post_material_assign,pre_loop_currents,pre_mesh_proximal,post_mesh_proximal,pre_solve}]
             [-w WAIT_FOR_LICENSE] [-P {cuff_only,nerve_only}]
             [-E {overwrite,error,selective}] [-e] [-r] [-S] [-c | -C]
             run_indices [run_indices ...]
```

Positional Arguments

run_indices	Space separated indices to run the pipeline over

Named Arguments

-b, --break-point	Possible choices: pre_geom_run, post_geom_run, pre_java, post_mesh_distal, pre_mesh_distal, post_material_assign, pre_loop_currents, pre_mesh_proximal, post_mesh_proximal, pre_solve Point in pipeline to exit and continue to next run
-w, --wait-for-license	Wait the specified number of hours for a comsol license to become available.
-P, --partial-fem	Possible choices: cuff_only, nerve_only Only generate the specified geometry.
-E, --export-behavior	Possible choices: overwrite, error, selective Behavior if n_sim export encounters extant data. Default is selective.
-e, --endo-only-solution	Store basis solutions for endoneurial geometry ONLY Default: False
-r, --render-deform	Pop-up window will render deformation operations Default: False
-S, --auto-submit	Automatically submit fibers after each run Default: False
-c, --comsol-progress	Print COMSOL progress to stdout Default: False
-C, --comsol-progress-popup	Show COMSOL progress in a pop-up window Default: False

install

install ASCENT

```
run install [-h] [--no-conda]
```

Named Arguments

--no-conda	Skip conda portion of installation Default: False

env_setup

Set ASCENT environment variables

```
run env_setup [-h]
```

clean_samples

Remove all files except those specified from Sample directories

```
run clean_samples [-h] sample_indices [sample_indices ...]
```

Positional Arguments

sample_indices	Space separated sample indices to clean

import_n_sims

Move NEURON outputs into ASCENT directories for analysis

```
run import_n_sims [-h] [-f | -D] run_indices [run_indices ...]
```

Positional Arguments

run_indices	Space separated run indices to import

Named Arguments

-f, --force	Import n_sims even if all thresholds are not found Default: False
-D, --delete-nsims	After importing delete n_sim folder from NSIM_EXPORT_PATH Default: False

mock_morphology_generator

Generate mock morpology for an ASCENT run

```
run mock_morphology_generator [-h] mock_sample_index
```

Positional Arguments

mock_sample_index	Mock Sample Index to generate

tidy_samples

Remove specified files from Sample directories

```
run tidy_samples [-h] sample_indices [sample_indices ...]
```

Positional Arguments

sample_indices	Space separated sample indices to tidy

2.2.4 submit.py

ASCENT: Automated Simulations to Characterize Electrical Nerve Thresholds

```
usage: submit.py [-h] [-p PARTITION] [-n NUM_CPU] [-m JOB_MEM] [-j NUM_JOBS]
                 [-l] [-A] [-s] [-S SLURM_PARAMS] [-L | -C] [-v]
                 [run_indices [run_indices ...]]
```

Positional Arguments

run_indices	Space separated indices to submit NEURON sims for

Named Arguments

-p, --partition	If submitting on a cluster, overrides slurm_params.json
-n, --num-cpu	For local submission: set number of CPUs to use, overrides run.json
-m, --job-mem	For cluster submission: set amount of RAM per job (in MB), overrides slurm_params.json
-j, --num-jobs	For cluster submission: set number of jobs per array, overrides slurm_params.json
-l, --list-runs	List info for available runs.z If supplying this argument, do not pass any run indices
-A, --all-runs	Submit all runs in the present export folder. If supplying this argument, do not pass any run indices Default: False
-s, --skip-summary	Begin submitting fibers without asking for confirmation Default: False
-S, --slurm-params	For cluster submission: string for additional slurm parameters (enclose in quotes)
-L, --local-submit	Set submission context to local, overrides run.json Default: False

-C, --cluster-submit	Set submission context to cluster, overrides run.json Default: False
-v, --verbose	Print detailed submission info Default: False

2.3 Information on ASCENT controls

2.3.1 Morphology Input Files

Each mask must be binary (i.e., white pixels ('1') for the segmented tissue and black pixels ('0') elsewhere) and must use Tagged Image File Format (i.e., `.tif`, or `.tiff`). All masks must be defined within the same field of view, be the same size, and be the same resolution. To convert between pixels of the input masks to dimensioned length (micrometers), the user must specify a `"ScaleInputMode"` in ***Sample*** (*JSON Configuration Files*). If using the mask input mode, a mask for the scale bar (`s.tif`) of known length (oriented horizontally) must be provided (see "Scale Bar" in Fig 2) and the length of the scale bar must be indicated in ***Sample*** (*JSON Configuration Files*). If using the ratio input mode, the user explicitly specifies the micrometers/pixel of the input masks in ***Sample*** (*JSON Configuration Files*), and no scale bar image is required.

The user is required to set the `"MaskInputMode"` in ***Sample*** (`"mask_input"`) to communicate the contents of the segmented histology files (*JSON Configuration Files*). Ideally, segmented images of boundaries for both the "outers" (`o.tif`) and "inners" (`i.tif`) of the perineurium will be provided, either as two separate files (`o.tif` and `i.tif`) or combined in the same image (`c.tif`) (see "Inners", "Outers", and "Combined" in Fig 2). However, if only inners are provided—which identify the outer edge of the endoneurium—a surrounding perineurium thickness is defined by the `"PerineuriumThicknessMode"` in ***Sample*** (`"ci_perineurium_thickness"`); the thickness is user-defined, relating perineurium thickness to features of the inners (e.g., their diameter). It should be noted that defining nerve morphology with only inners does not allow the model to represent accurately fascicles containing multiple endoneurium inners within a single outer perineurium boundary ("peanut" fascicles; see an example in Fig 2); in this case, each inner trace will be assumed to represent a single independent fascicle that does not share its perineurium with any other inners; more accurate representation requires segmentation of the "outers" as well.

The user is required to set the `"NerveMode"` in ***Sample*** ("nerve") to communicate the contents of the segmented histology files (*JSON Configuration Files*). The outer nerve boundary, if present, is defined with a separate mask (`n.tif`). In the case of a compound nerve with epineurium, the pipeline expects the outer boundary of the epineurium to be provided as the "nerve". In the case of a nerve with a single fascicle, no nerve mask is required—in which case either the outer perineurium boundary (if present) or the inner perineurium boundary (otherwise) is used as the nerve boundary—although one may be provided if epineurium or other tissue that would be within the cuff is present in the sample histology.

Lastly, an "orientation" mask (`a.tif`) can be optionally defined. This mask should be all black except for a small portion that is white, representing the position to which the cuff must be rotated. The angle is measured *relative to the centroid of the nerve/singular fascicle*, so this image should be constructed while referencing `n.tif` (or, if monofascicular, `i.tif`, `o.tif`, or `c.tif`). By default, the 0º position of our cuffs correspond with the coordinate halfway along the arc length of the cuff inner diameter (i.e., the cuff will be rotated such that the sample center, cuff contact center, and centroid of the white portion of `a.tif` form a line) while the circular portion of a cuff's diameter is centered at the origin (Note: this rotation process uses `"angle_to_contacts_deg"` and `"fixed_point"` in a "preset" cuff's JSON file, see *Creating Custom Cuffs* and *Cuff Placement on the Nerve*). If `a.tif` is provided, other cuff rotation methods (`"cuff_shift"` in ***Model***, which calculate `"pos_ang"`) are overridden.

The user must provide segmented image morphology files, either from histology or the `mock_morphology_generator.py` script, with a specific naming convention in the `input/` directory.

- Raw RGB image, to be available for convenience and used for data visualization: `r.tif` (optional).

- Combined (i.e., inners and outers): `c.tif`.
- Inners: `i.tif`
  - An "inner" is the internal boundary of the perineurium that forms the boundary between the perineurium and the endoneurium.
- Outers: `o.tif`
  - An "outer" is the external boundary of the perineurium that forms the boundary between the perineurium and the epineurium or extraneural medium.
- Scale bar: `s.tif` (scale bar oriented horizontally, required unless scale input mode is set to ratio).
- Nerve: `n.tif` (optional for monofascicular nerves).
- Orientation: `a.tif` (optional).

For an example of input files, see Fig 2. The user must properly set the `"MaskInputMode"` in ***Sample*** (`"mask_input"`) for their provided segmented image morphology files (*JSON Configuration Files*).

2.3.2 Control of medium surrounding nerve and cuff electrode

The medium surrounding the nerve and cuff electrode (e.g., fat, skeletal muscle) must contain a "proximal" domain, which runs the full length of the nerve, and may optionally include a "distal" domain. The parameterization for the geometry of the "proximal" and "distal" domains is shown below in Figure A. For details on how to define the "proximal" and "distal" domain geometries and meshing parameters, see *Model Parameters*.

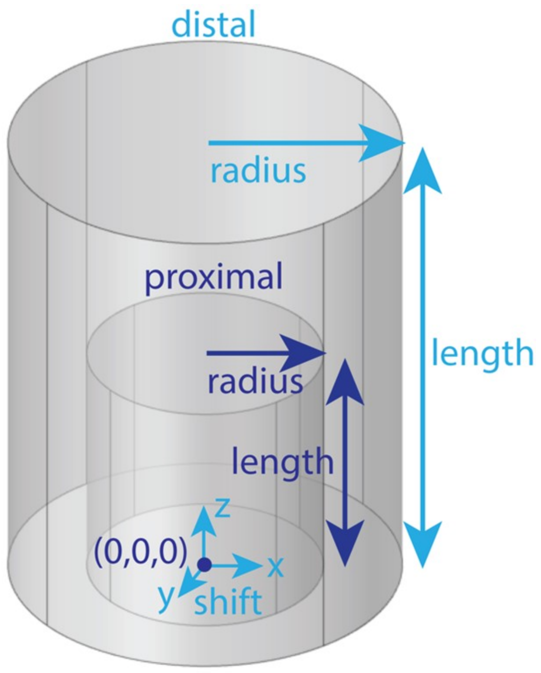


Figure A. The user must define a "proximal" domain, and may optionally define a "distal" domain for independent assignment of meshing parameters for the site of stimulation from the rest of the FEM. The "proximal" domain runs the full length of the nerve and is anchored at (0,0,0). The distal domain's radius and length may be independently assigned, and the entire distal domain may be shifted ("shift": (x, y, z)). Having a proximal domain that is overly voluminous can significantly decrease COMSOL meshing efficiency and even, rarely, cause errors. At all costs, avoid having a proximal or distal domain whose boundary intersects with a geometry (other than the nerve ends, which are by definition at the longitudinal boundaries of the proximal domain) or the boundary of other geometries (e.g., the cuff-nerve boundary); this will likely create a meshing error.

2.3.3 Cuff placement on nerve

This section provides an overview of how the cuff is placed on the nerve. The `compute_cuff_shift()` method within Runner (`src/runner.py`) determines the cuff's rotation around the nerve and translation in the (x,y)-plane. The pipeline imports the coordinates of the traces for the nerve tissue boundaries saved in `samples/<sample_index>/slides/0/0/sectionwise2d/` which are, by convention, shifted such that the centroid of the nerve is at the origin (0,0) (i.e., nerve centroid from best-fit ellipse if nerve trace (`n.tif`) is provided, inner or outer best-fit ellipse centroid for monofascicular nerves without nerve trace). Importantly, the nerve sample cross section is never moved from or rotated around the origin in COMSOL. By maintaining consistent nerve location across all ***Model's*** for a ***Sample,*** the coordinates in `fibersets/` are correct for any orientation of a cuff on the nerve.

ASCENT has different `CuffShiftModes` (i.e., `"cuff_shift"` parameter in ***Model***) that control the translation of the cuff (i.e., "shift" JSON Object in ***Model***) and default rotation around the nerve (i.e., `"pos_ang"` in ***Model***). Runner's `compute_cuff_shift()` method is easily expandable for users to add their own `CuffShiftModes` to control cuff placement on the nerve.

The rotation and translation of the cuff are populated automatically by the `compute_cuff_shift()` method based on sample morphology, parameterization of the "preset" cuff, and the `CuffShiftMode`, and are defined in the "cuff" JSON Object ("shift" and "rotate") in ***Model***.

For "naïve" `CuffShiftModes` (i.e., `"NAIVE_ROTATION_MIN_CIRCLE_BOUNDARY"`, `"NAIVE_ROTATION_TRACE_BOUNDARY"`) the cuff is placed on the nerve with rotation according to the parameters used to instantiate the cuff from part primitives (*Part Primitives and Custom Cuffs*). If the user would like to rotate the cuff from beyond this position, they may set the `"add_ang"` parameter in ***Model*** (*Model Parameters*). For naïve `CuffShiftModes`, the cuff is shifted along the vector from (0,0) in the direction of the `"angle_to_contacts_deg"` parameter in the "preset" JSON file. `"NAIVE_ROTATION_MIN_CIRCLE_BOUNDARY"` `CuffShiftMode` moves the cuff toward the nerve until the nerve's minimum radius enclosing circle is within the distance of the `"thk_medium_gap_internal"` parameter for the cuff. `"NAIVE_ROTATION_TRACE_BOUNDARY"` `CuffShiftMode` moves the cuff toward the nerve until the nerve's outermost Trace (i.e., for monofascicular nerve an inner or outer, and same result as `"NAIVE_ROTATION_MIN_CIRCLE_BOUNDARY"` for nerve's with epineurium) is within the distance of the `"thk_medium_gap_internal"` parameter for the cuff. Note: orientation masks (`a.tif`) are ignored when using these modes.

For "automatic" `CuffShiftModes` (i.e., `"AUTO_ROTATION_MIN_CIRCLE_BOUNDARY"`, `"AUTO_ROTATION_TRACE_BOUNDARY"`) the cuff is rotated around the nerve based on the size and position of the nerve's fascicle(s) before the cuff is moved toward the nerve sample (Figure A). The point at the intersection of the vector from (0,0) in the direction of the `"angle_to_contacts_deg"` parameter in the "preset" JSON file with the cuff (i.e., cuff's "center" in following text) is rotated to meet a specific location of the nerve/monofascicle's surface. Specifically, the center of the cuff is rotated around the nerve to make (0,0), the center of the cuff, and ***Sample's*** `"fascicle_centroid"` (computed with Slide's `fascicle_centroid()` method, which calculates the area and centroid of each inner and then averages the inners' centroids weighted by each inner's area) colinear. If the user would like to rotate the cuff from beyond this position, they may set the `"add_ang"` parameter in ***Model*** (*Model Parameters*). The user may override the default "AUTO" rotation of the cuff on the nerve by adding an orientation mask (`a.tif`) to align a certain surface of the nerve sample with the cuff's center (../Running_ASCENT/Info.md#morphology-Input-Files)). This behavior depends on the cuff's `"fixed_point"` parameter (*Creating Custom Cuffs*) The `"AUTO_ROTATION_MIN_CIRCLE_BOUNDARY"` `CuffShiftMode` moves the cuff toward the nerve until the nerve's minimum radius enclosing circle is within the distance of the `"thk_medium_gap_internal"` parameter for the cuff. `"AUTO_ROTATION_TRACE_BOUNDARY"` `CuffShiftMode` moves the cuff toward the nerve until the nerve's outermost Trace (i.e., for monofascicular nerve an inner or outer, and same result as `"AUTO_ROTATION_MIN_CIRCLE_BOUNDARY"` for nerve's with epineurium) is within the distance of the `"thk_medium_gap_internal"` parameter for the cuff.

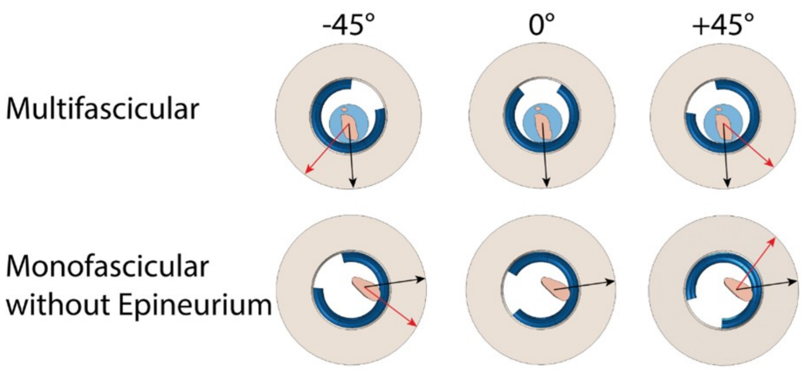


Figure A. Demonstration of cuff placement on a multifascicular nerve (top) and a monofascicular nerve without epineurium (bottom) with the same "preset" cuff (Purdue.json) for three different cuff rotations using the "AUTO_ROTATION_TRACE_BOUNDARY" CuffShiftMode. The cuff rotations are different in the top and bottom rows since the point on the surface of the nerve sample closest to the most endoneurium is unique to each sample (black arrows). Additional angles of rotation were applied to the cuff directly using the "add_ang" parameter in the ***Model's*** "cuff" JSON Object (red arrows).

The default z-position of each part along the nerve is defined in the "preset" cuff JSON file by the expression assigned to the part instance's "Center" parameter (referenced to z = 0 at one end of the model's proximal cylindrical domain). However, if the user would like to move the entire "preset" cuff along the length of the nerve, in the "cuff" JSON Object within ***Model***, the user may change the "z" parameter.

Since some cuffs can open in response to a nerve diameter larger than the manufactured cuff's inner diameter, they maybe be parameterized as a function of `"R_in"`. In this case, in the "preset" cuff JSON, the "expandable" Boolean parameter is true. If the cuff is "expandable" and the minimum enclosing diameter of the sample is larger than the cuff, the program will modify the angle for the center of the cuff to preserve the length of materials. If the cuff is not expandable and the sum of the minimum enclosing circle radius of the sample and `"thk_medium_gap_internal"` are larger than the inner radius of the cuff, an error is thrown as the cuff cannot accommodate the sample morphology.

The inner radius of the cuff is defined within the list of "params" in each "preset" cuff configuration file as `"R_in"`, and a JSON Object called "offset" contains a parameterized definition of any additional buffer required within the inner radius of the cuff (e.g., exposed wire contacts as in the `Purdue.json` cuff "preset"). Each key in the "offset" JSON Object must match a parameter value defined in the "params" list of the cuff configuration file. For each key in "offset", the value is the multiplicative coefficient for the parameter key to include in a sum of all key-value products. For example, in `Purdue.json`:

```
"offset": {
  "sep_wire_P": 1, // separation between outer boundary of wire contact and internal
                 // surface of insulator
  "r_wire_P": 2 // radius of the wire contact's gauge
}
```

This JSON Object in `Purdue.json` will instruct the system to maintain added separation between the internal surface of the cuff and the nerve of:

The div element has its own alignment attribute, align.

2.3.4 Simulation Protocols

Fiber response to electrical stimulation is computed by applying electric potentials along the length of a fiber from COMSOL as a time-varying signal in NEURON. The stimulation waveform, saved in a `n_sim's data/inputs/` directory as `waveform.dat`, is unscaled (i.e., the maximum current magnitude at any timestep is +/-1), and is then scaled by the current amplitude in `RunSim.hoc` to either simulate fiber thresholds of activation or block with a binary search algorithm, or response to set amplitudes.

Binary search

In searching for activation thresholds (i.e., the minimum stimulation amplitude required to generate a propagating action potential) or block thresholds (i.e., the minimum stimulation amplitude required to block the propagation of an action potential) in the pipeline, the NEURON code uses a binary search algorithm.

The basics of a binary search algorithm are as follows. By starting with one value that is above threshold (i.e., upper bound) and one value that is below threshold (i.e., lower bound), the program tests the midpoint amplitude to determine if it is above or below threshold. If the midpoint amplitude is found to be below threshold, the midpoint amplitude becomes the new lower bound. However, if the midpoint amplitude is found to be above threshold, the midpoint amplitude becomes the new upper bound. At each iteration of this process, half of the remaining amplitude range is removed. The process is continued until the termination criteria is satisfied (e.g., some threshold resolution tolerance is achieved). The average performance of a binary search algorithm is $(\log(n))$ where n is the number of elements in the search array (i.e., linearly spaced range of amplitudes).

In the pipeline, the binary search protocol parameters (i.e., activation or block criteria, threshold criteria, method for searching for starting upper- and lower bounds, or termination criteria) are contained in the "protocol" JSON Object within ***Sim*** (*Sim Parameters*).

Activation threshold protocol

The pipeline has a NEURON simulation protocol for determining thresholds of activation of nerve fibers in response to extracellular stimulation. Threshold amplitude for fiber activation is defined as the minimum stimulation amplitude required to initiate a propagating action potential. The pipeline uses a binary search algorithm to converge on the threshold amplitude. Current amplitudes are determined to be above threshold if the stimulation results in at least `n_AP` propagating action potentials detected at 75% of the fiber's length (note: location can be specified by user with `"ap_detect_location"` parameter in ***Sim***) (*Sim Parameters*). The parameters for control over the activation threshold protocol are found in ***Sim*** within the "protocol" JSON Object (*Sim Parameters*).

Block threshold protocol

The pipeline has a NEURON simulation protocol for determining block thresholds for nerve fibers in response to extracellular stimulation. Threshold amplitude for fiber block is defined as the minimum stimulation amplitude required to block a propagating action potential. The simulation protocol for determining block thresholds starts by delivering the blocking waveform through the cuff. After a user-defined delay during the stimulation onset period, the protocol delivers a test pulse (or a train of pulses if the user chooses) where the user placed it (see "ind" parameter in ***Sim*** within the `"intracellular_stim"` JSON Object (*Sim Parameters*)), near the proximal end. The code checks for action potentials near the distal end of the fiber (see `"ap_detect_location"` parameter in ***Sim*** within the "threshold" JSON Object (*Sim Parameters*)). If at least one action potential is detected, then transmission of the test pulse occurred (i.e., the stimulation amplitude is below block threshold). However, the absence of an action potential indicates block (i.e., the stimulation amplitude is above block threshold). The pipeline uses a binary search algorithm to converge on the threshold amplitude. The parameters for control over the block threshold protocol are found in ***Sim*** within the "protocol" JSON Object (*Sim Parameters*).

The user must be careful in setting the initial upper and lower bounds of the binary search for block thresholds. Especially for small diameter myelinated fibers, users must be aware of and check for re-excitation using a stimulation amplitude sweep [Pelot *et al.*, 2017].

Response to set amplitudes

Alternatively, users may simulate the response of nerve fibers in response to extracellular stimulation for a user-specified set of amplitudes. The "protocol" JSON Object within ***Sim*** contains the set of amplitudes that the user would like to simulate (*Sim Parameters*).

2.3.5 Implementation of NEURON fiber models

Myelinated fiber models

The `CreateAxon_Myel.hoc` file is loaded in `Wrapper.hoc` if the user chooses either `"MRG_DISCRETE"` or `"MRG_INTERPOLATION"`. The length of each section in NEURON varies depending on both the diameter and the "FiberGeometry" mode chosen in ***Sim***.

MRG discrete diameter (as previously published)

The "FiberGeometry" mode `"MRG_DISCRETE"` in ***Sim*** instructs the program to simulate a double cable structure for mammalian myelinated fibers [McIntyre *et al.*, 2004, McIntyre *et al.*, 2002]. In the pipeline, we refer to this model as `"MRG_DISCRETE"` since the model's geometric parameters were originally published for a *discrete* list of fiber diameters: 1, 2, 5.7, 7.3, 8.7, 10, 11.5, 12.8, 14.0, 15.0, and 16.0 m. Since the MRG fiber model has distinct geometric dimensions for each fiber diameter, the parameters are stored in `config/system/fiber_z.json` as lists in the `"MRG_DISCRETE"` JSON Object, where a value's index corresponds to the index of the discrete diameter in "diameters". The parameters are used by the Fiberset class to create `fibersets/` (i.e., coordinates to probe `potentials/` from COMSOL) for MRG fibers.

MRG interpolated diameters

The `"FiberGeometry"` mode `"MRG_INTERPOLATION"` in ***Sim*** instructs the program to simulate a double cable structure for mammalian myelinated fibers for any diameter fiber between 2 and 16 µm (throws an error if not in this range) by using an *interpolation* over the originally published fiber geometries [McIntyre *et al.*, 2004, McIntyre *et al.*, 2002]. In the pipeline, we refer to this model as `"MRG_INTERPOLATION"` since it enables the user to simulate any fiber diameter between the originally published diameters.

The parameters in the `"MRG_INTERPOLATION"` JSON Object in `config/system/fiber_z.json` are used by the Fiberset class to create `fibersets/` (i.e., coordinates at which to sample `potentials/` from COMSOL) for interpolated MRG fibers. Since the parameter values relate to fiber "diameter" as a continuous variable, the expressions for all the dimensions that change with fiber diameter, as shown in Figure A, are stored as a String that is computed using Python's built-in `"eval()"` function.

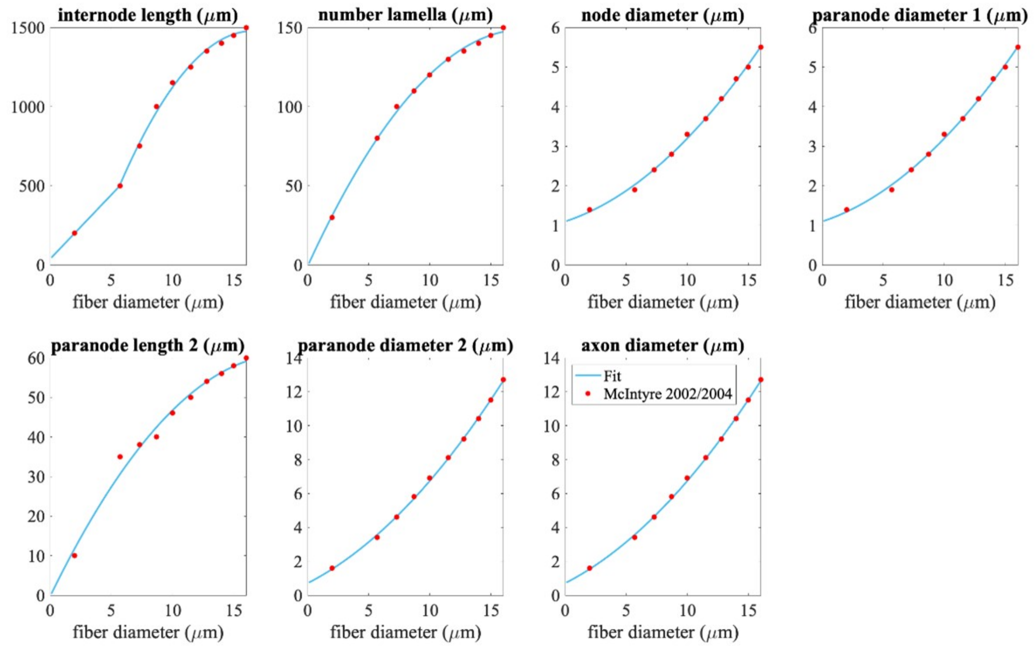


Figure A. Piecewise polynomial fits to published MRG fiber parameters. Single quadratic fits were used for all parameters except for internode length, which has a linear fit below 5.643 µm (using MRG data at 2 and 5.7 µm) and a single quadratic fit at diameters greater than or equal to 5.643 µm (using MRG data >= 5.7 µm); 5.643 µm is the fiber diameter at which the linear and quadratic fits intersected. The fiber diameter is the diameter of the myelin. "Paranode 1" is the MYSA section, "paranode 2" is the FLUT section, and "internode" is the STIN section. The axon diameter is the same for the node of Ranvier and MYSA ("node diameter"), as well as for the FLUT and STIN ("axon diameter"). The node and MYSA lengths are fixed at 1 and 3 m, respectively, for all fiber diameters.

We compared fiber activation thresholds between the originally published MRG fiber models and the interpolated MRG ultrastructure (evaluated at the original diameters) at a single location in a rat cervical vagus nerve stimulated with the bipolar Purdue cuff. Each fiber was placed at the centroid of the best-fit ellipse of the monofascicular nerve sample. The waveform was a single biphasic pulse using `"BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW"` with 100 µs for the first phase, 100 µs interphase (0 mA), and 400 µs for the second phase (cathodic/anodic at one contact and anodic/cathodic at the other contact). The thresholds between the originally published models and the interpolation of the MRG fiber diameters are compared in Figure B below. The threshold values were determined using a binary search until the upper and lower bound stimulation amplitudes were within 1%.

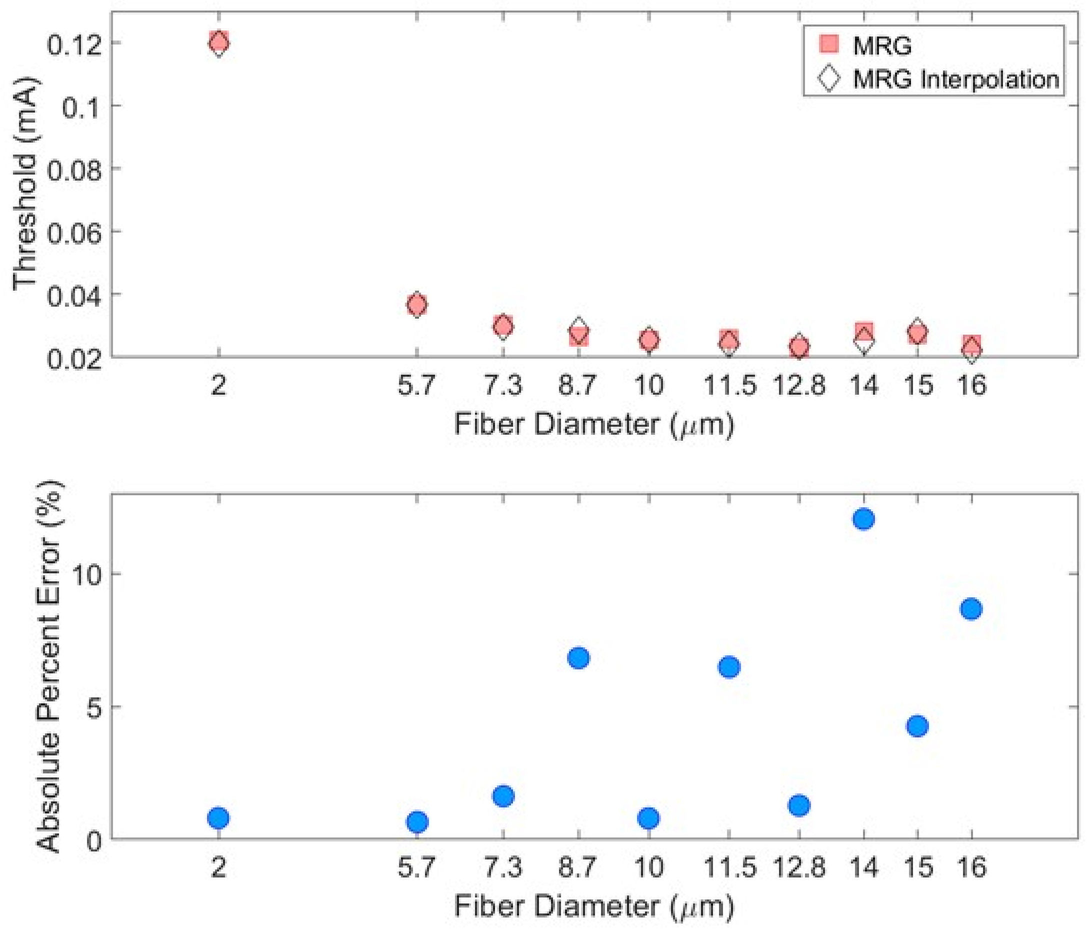


Figure B. Comparison of thresholds between the originally published models and the interpolation of the MRG fiber diameters (evaluated at the original diameters). Thresholds are expected to vary between the originally published models and the interpolated fiber geometries given their slightly different ultrastructure parameters (Figure A). Used original MRG thresholds as reference.

Unmyelinated Fiber Models

The pipeline includes several unmyelinated (i.e., C-fiber) models [Rattay and Aberham, 1993, Sundt *et al.*, 2015, Tigerholm *et al.*, 2014]. Users should be aware of the `"delta_zs"` parameter that they are using in `config/system/fiber_z.json`, which controls the spatial discretization of the fiber (i.e., the length of each section).

2.3.6 Defining and assigning materials in COMSOL

Materials are defined in the COMSOL "Materials" node for each material "function" indicated in the "preset" cuff configuration file (i.e., cuff "insulator", contact "conductor", contact "recess", and cuff "fill") and nerve domain (i.e., endoneurium, perineurium, epineurium). Material properties for each function are assigned in ***Model's*** "conductivities" JSON Object by either referencing materials in the default materials library (`config/system/materials.json`) by name, or with explicit definitions of a materials name and conductivity as a JSON Object (*Material Parameters*). See [link](link to ModelWrapper.addMaterialDefinitions()) for code one how this happens.

Adding and assigning default material properties

Default material properties defined in `config/system/materials.json` are listed in Table A. To accommodate automation of frequency-dependent material properties (for a single frequency, i.e., sinusoid), parameters for material conductivity that are dependent on the stimulation frequency are calculated in Runner's `compute_electrical_parameters()` method and saved to ***Model*** before the handoff() method is called. Our pipeline supports calculation of the frequency-dependent conductivity of the perineurium based on measurements from the frog sciatic nerve [Weerasuriya *et al.*, 1984] using the `rho_weerasuriya()` method in the Python Waveform class. See Fig. 2 for identification of tissue types in a compound nerve cross section (i.e., epineurium, perineurium, endoneurium).

Table A. Default material conductivities.

Material	Conductivity	References
silicone	10^-12 [S/m]	[Callister and Rethwisch, 2011]
platinum	9.43 10^6 [S/m]	[de Podesta, 1997]
endoneurium	{1/6, 1/6, 1/1.75} [S/m]	[Jr. and BeMent, 1965, Pelot *et al.*, 2018]
epineurium	1/6.3 [S/m]	[Grill and Mortimer, 1994, Pelot *et al.*, 2017, Stolinski, 1995]
muscle	{0.086, 0.086, 0.35} [S/m]	[Gielen *et al.*, 1984]
fat	1/30 [S/m]	[Geddes and Baker, 1967]
encapsulation	1/6.3 [S/m]	[Grill and Mortimer, 1994]
saline	1.76 [S/m]	[Horch and Kipke, 2017]
perineurium	1/1149 [S/m]	[Pelot *et al.*, 2018, Weerasuriya *et al.*, 1984]

Definition of perineurium

The perineurium is a thin highly resistive layer of connective tissue and has a profound impact on thresholds of activation and block. Our previous modeling work demonstrates that representing the perineurium with a thin layer approximation (Rm = rho*peri_thk), rather than as a thinly meshed domain, reduces mesh complexity and is a reasonable approximation [Pelot *et al.*, 2018]. Therefore, perineurium can be modeled with a thin layer approximation (except with "peanut" fascicles; see an example in Fig 2), termed "contact impedance" in COMSOL (if ***Model's*** `"use_ci"` parameter is true (*Model Parameters*)), which relates the normal component of the current density through the surface to the drop in electric potentials and the sheet resistance :

The sheet resistance (ρ_s) is defined as the sheet thickness divided by the material bulk conductivity :

Our previously published work quantified the relationship between fascicle diameter and perineurium thickness [Pelot *et al.*, 2020] (Table A).

Table A. Previously published relationships between fascicle diameter and perineurium thickness.

Species	**peri_thk: *f*(species, dfasc)**	**References**
Rat	peri_thk = 0.01292*dfasc + 1.367 [um]	[Pelot *et al.*, 2020]
Pig	peri_thk = 0.02547*dfasc + 3.440 [um]	[Pelot *et al.*, 2020]
Human	peri_thk = 0.03702*dfasc + 10.50 [um]	[Pelot *et al.*, 2020]

The “rho_perineurium” parameter in ***Model*** can take either of two modes:

- “RHO_WEERASURIYA”: The perineurium conductivity value changes with the frequency of electrical stimulation (for a single value, not a spectrum, defined in ***Model*** as “frequency”) and temperature (using a Q10 adjustment, defined in ***Model*** as “temperature”) based on measurements of frog sciatic perineurium [Pelot *et al.*, 2018, Weerasuriya *et al.*, 1984]. The equation is defined in `src/core/Waveform.py` in the `rho_weerasuriya()` method.
- “MANUAL”: Conductivity value assigned to the perineurium is as explicitly defined in either `materials.json` or ***Model*** without any corrections for temperature or frequency.

CHAPTER

THREE

JSON CONFIGURATION FILES

3.1 JSON Configuration File Overview

We store parameters in JSON configuration files because the JSON format is accessible, readable, and well-documented for metadata interchange. The configuration files inform the pipeline in its operations and provide a traceable history of the parameters used to generate data.

For each JSON file, we provide a brief overview, a statement of where the file must be placed in the directory structure, and a description of its contents. For a detailed description of each JSON file (i.e., which parameters are required or optional, known value data types, and known values), see *JSON Parameters*. Though JSON does not allow comments, users may want to add notes to a JSON file (e.g., to remember what a ***Sample***, ***Model***, or ***Sim*** file was used to accomplish). The user can simply add a key to the JSON file that is not in use (e.g., "notes") and provide a value (`String`) with a message.

3.1.1 User configuration files

run.json

The `run.json` file is passed to `pipeline.py` to instruct the program which ***Sample***, ***Model(s)***, and ***Sim(s)*** to run. All `run.json` files are stored in the `config/user/runs/` directory. Since each ***Sample***, ***Model***, and ***Sim*** is indexed, their indices are used as the identifying values in `run.json`. Additionally, the file contains break points that enable the user to terminate the pipeline at intermediate processes, flags for user control over which COMSOL files are saved, flags to selectively build a `"debug_geom.mph"` file of just the nerve or cuff electrode, and flags to recycle previous meshes where appropriate. Lastly, the `run.json` file reports which ***Model*** indices were generated successfully in COMSOL and indicates if the user is submitting the NEURON jobs to a SLURM cluster or locally.

sample.json

An example ***Sample*** configuration file is stored in `config/templates/` for users to reference when defining their own input nerve morphology from histology or from the mock nerve morphology generator *Mock Morphology*. A user's `sample.json` file is saved in the `samples/<sample_index>/` directory. The file contains information about the sample's properties and how to process the nerve morphology before placing the nerve in a cuff electrode. The pipeline's processes informed by ***Sample*** output the Python object `sample.obj`.

The information in ***Sample*** must parallel how the sample morphology binary images are saved on the user's machine (sample name, path to the sample). The user must define the `"mask_input"` parameter in ***Sample*** to indicate which set of input masks will be used. ***Sample*** also contains parameters that determine how the morphology is processed before being represented in the FEM such as shrinkage correction ("shrinkage"), required minimum fascicle separation (`"boundary_separation"`), presence of epineurium ("nerve"), perineurium thickness (`"ci_perineurium_thickness"`), deformation method ("deform"), reshaped nerve profile (`"reshape_nerve"`), and parameters for specifying CAD geometry input file formats for nerve morphology ("write"). ***Sample*** also contains information about the mask scaling (i.e. `"scale_bar_length"` if supplying a binary image of a scale bar or, if no scale bar image, `"scale_ratio"`).

The value of the `"ci_perineurium_thickness"` in ***Sample*** refers to a JSON Object in `config/system/ci_peri_thickness.json` that contains coefficients for linear relationships between inner diameter and perineurium thickness (i.e., thkperi,inner = a*(diameterinner) + b). In `ci_peri_thickness.json`, we provided a `"PerineuriumThicknessMode"` named `"GRINBERG_2008"`, which defines perineurium thickness as 3% of inner diameter [Grinberg *et al.*, 2008], and relationships for human, pig, and rat vagus nerve perineurium thickness (i.e., `"HUMAN_VN_INHOUSE_200601"`, `"PIG_VN_INHOUSE_200523"`, and `"RAT_VN_INHOUSE_200601"`) [Pelot *et al.*, 2020]. As additional vagus nerve morphometry data become available, users may define perineurium thickness with new models by adding the coefficients to this JSON file.***

mock_sample.json

The `mock_sample.json` file, which is stored in the file structure in `config/user/mock_samples/<mock_sample_index>/`, is used to define binary segmented images that serve as inputs to the pipeline. In the "populate" JSON Object, the user must define the "PopulateMode" (e.g., EXPLICIT, TRUNCNORM, UNIFORM defined by the "mode" parameter), which defines the process by which the nerve morphology is defined in the MockSample Python class. Each `"PopulateMode"` requires a certain set of parameters to define the nerve and to define and place the fascicles; the set of parameters for each `"PopulateMode"` are defined in `config/templates/mock_sample_params_all_modes.json`.

Probabilistic "PopulateModes" (i.e., TRUNCNORM, UNIFORM) populate an elliptical nerve with elliptical fascicles of diameter and eccentricity defined by a statistical distribution. Since the nerve morphology parameters are defined probabilistically, a "seed" parameter is required for the random number generator to enable reproducibility. The fascicles are placed at randomly chosen locations within the nerve using a disk point picking method; the fascicles are placed at a rotational orientation randomly chosen from 0-360o. If a fascicle is placed in the nerve without maintaining a user-defined `"min_fascicle_separation"` distance from the other fascicles and the nerve, another randomly chosen point within the nerve is chosen until either a location that preserves a minimum separation is achieved or the program exceeds a maximum number of attempts (`"max_attempt_iter"`).

The EXPLICIT `"PopulateMode"` populates an elliptical nerve with elliptical fascicles of user-defined sizes, locations, and rotations. The program validates that the defined fascicle ellipses are at least `"min_fascicle_separation"` distance apart; otherwise, if the conditions are not met, the program throws an error.

model.json

An example ***Model*** configuration file is stored in `config/templates/` for users to reference when creating their own FEMs. As such, `model.json`, which is stored in the file structure in `samples/<sample_index>/models/<model_index>/`, contains information to define an FEM uniquely. ***Model*** defines the cuff electrode geometry and positioning, the simulated environment (e.g., surrounding medium dimensions, material properties (including temperature and frequency factors of material conductivity), and physics), the meshing parameters (i.e., how the volume is discretized), and output statistics (e.g., time required to mesh, mesh element quality measures).

sim.json

An example ***Sim*** configuration file is stored in `config/templates/` for users to reference when creating their own simulations of fiber responses to stimulation for a sample in a FEM. All simulation configuration files are stored in the `config/user/sims/` directory. ***Sim*** defines fiber types, fiber locations in the FEM, fiber length, extracellular (e.g., pulse repetition frequency) and intracellular stimulation, and input parameters to NEURON (e.g., parameters to be saved in the output, binary search algorithm bounds and resolution). Since users may want to sweep parameters at the ***Sim*** configuration level (e.g., fiber types, fiber locations, waveforms), a pared down copy of ***Sim*** that contains a single value for each parameter (rather than a list) is saved within the corresponding `n_sims/` directory (*Sim Parameters*). These pared down files are provided for convenience, so that the user can inspect which parameters were used in a single NEURON simulation, and they do not hold any other function within the pipeline.

query_criteria.json

In data analysis, summary, and plotting, the user needs to inform the program which output data are of interest. The `query_criteria.json` file stores the "criteria" for a user's search through previously processed data. The `query_criteria.json` file may be used to guide the Query class's searching algorithm in the `run()` method. We suggest that all `query_criteria.json`-like files are stored in the `config/user/query_criteria/` directory; however, the location of these files is arbitrary, and when initializing the Query object, the user must manually pass in the path of either the `query_criteria.json`-like file or the hard-coded criteria as a Python dictionary. An instance of the Query class contains the "criteria" and an empty `_result`, which is populated by Query's `run()` method with found indices of ***Sample***, ***Model***, and ***Sim*** that match the criteria given.

Query's `run()` method loops through all provided indices (i.e., ***Sample***, ***Model***, ***Sim***) in the query criteria, and calls `_match()` when a possible match is found. Note that the presence of an underscore in the `_match()` method name indicates that it is for internal use only (not to be called by external files). The `_match()` method compares its two inputs, (1) `query_criteria.json` and (2) either ***Sample*** (i.e., `sample.json`), ***Model*** (i.e., `model.json`), or ***Sim*** (`sim.json`); the two JSON files are loaded into memory as Python dictionaries. The method returns a Boolean indicating if the input configuration file satisfies the restricted parameter values defined in `query_criteria.json`. The user may explicitly specify the indices of the ***Sample***, ***Model***, and ***Sim*** configuration files of interest simultaneously with restricted criteria for specific parameter values. The indices added will be returned in addition to matches found from the search criteria in the ***Sample***, ***Model***, and ***Sim*** criteria JSON Objects.

The `query_criteria.json` file contains JSON Objects for each of the ***Sample***, ***Model***, and ***Sim*** configuration files. Within each JSON Object, parameter keys can be added with a desired value that must be matched in a query result. If the parameter of interest is found nested within a JSON Object structure or list in the configuration file, the same hierarchy must be replicated in the `query_criteria.json` file.

The `query_criteria.json` parameter `"partial_matches"` is a Boolean indicating whether the search should return indices of ***Sample***, ***Model***, and ***Sim*** configuration files that are a partial match, i.e., the parameters in `query_criteria.json` are satisfied by a subset of parameters listed in the found JSON configuration.

The `query_criteria.json` parameter `"include_downstream"` is a Boolean indicating whether the search should return indices of downstream (***Sample***>***Model***>***Sim***) configurations that exist if match criteria are not provided for them. For example, if only criteria for a ***Sample*** and ***Model*** are provided, Query will return the indices of ***Sample*** and ***Model*** that match the criteria. In addition, the indices of the ***Sims*** downstream of the matches are included in the result if `"include_downstream"` is true (since the user did not specify criteria for ***Sim***). Otherwise, if `"include_downstream"` is false, no ***Sim*** indices are returned.

3.1.2 System configuration files

Note: system configuration files are located in `config/system/`.

env.json

The `env.json` file stores the file paths for:

- COMSOL
- Java JDK
- The project path (i.e., the path to the root of the ASCENT pipeline)
- Destination directory for NEURON simulations to run (this could be the directory from which the user calls NEURON, or an intermediate directory from which the user will move the files to a computer cluster)

When the pipeline is run, the key-value pairs are stored as environment variables so that they are globally accessible.

exceptions.json

The `exceptions.json` file contains a list of exceptions that are intentionally thrown in the Python portion of the pipeline. Each error has its own "code" (index), and "text" (informative message hinting to the reason the program failed). As developers add new methods to Python classes that inherit the Exceptionable class, appending errors onto `exceptions.json` that are called from Python code file (i.e., `self.throw(<exception index>)`) will help give informative feedback to the user.

materials.json

The `materials.json` file contains default values for material properties that can be assigned to each type of neural tissue, each electrode material, the extraneural medium, and the medium between the nerve and inner cuff surface. The materials are referenced by using their labels in the "conductivities" JSON Object of ***Model***.

fiber_z.json

The `fiber_z.json` file defines z-coordinates to be sampled along the length of the FEM for different fiber types to be simulated in NEURON. In some instances, the section lengths are a single fixed value. In other instances, such as the MRG model [McIntyre *et al.*, 2002], the section lengths are defined for each fiber diameter in a discrete list. Section lengths can also be a continuous function of a parameter, such as fiber diameter, defined as a mathematical relationship in the form of a string to be evaluated in Python. Additionally, the file contains instructions (e.g., flags) that corresponds to fiber-type specific operations in NEURON.

ci_perineurium_thickness.json

In the case of fascicles with exactly one inner perineurium trace for each outer perineurium trace, to reduce the required computational resources, the pipeline can represent the perineurium using a thin layer approximation in COMSOL (*Perineurium Properties*). Specifically, if ***Model's*** `"use_ci"` parameter is true, the perineurium is modeled as a surface with a sheet resistance (termed "contact impedance" in COMSOL) defined by the product of the resistivity and thickness. The thickness is calculated as half of the difference between the effective circular diameters of the outer and inner perineurium traces. If each fascicle is only defined by a single trace (rather than inner and outer perineurium traces), the user chooses from a list of modes in ***Sample*** for assigning a perineurium thickness (e.g., 3% of fascicle diameter [Grinberg *et al.*, 2008], `"ci_perineurium_thickness"` parameter in ***Sample***).

mesh_dependent_model.json

Since meshing can take substantial time and RAM, if the FEM has the same geometry and mesh parameters as a previous model, this JSON file allows the mesh to be reused if the `mesh.mph` file is saved. In `mesh_dependent_model.json`, the keys match those found in ***Model***, however, instead of containing parameter values, each key's value is a Boolean indicating true if the parameter value must match between two ***Model*** configurations to recycle a mesh, or false if a different parameter value would not prohibit a mesh be reused. The `mesh_dependent_model.json` file is used by our `ModelSearcher` Java utility class (*Java Utility Classes*).

3.2 JSON Configuration Parameter Guide

Notes:

See *JSON Overview* for a summary of the contents and usage of each of the following JSON files used in ASCENT.

`"//"` is not valid JSON syntax; comments are not possible in JSON. However, we sparingly used this notation in the JSON examples below to provide context or more information about the associated line. Each value following a key in the syntax denotes the *type* of the value, not its literal value: "[<Type X>, . . .]" syntax indicates that the type is an array of Type X. Occasionally, a single value may be substituted for the list if only a single value is desired, but this functionality differs between keys, so be sure to read the documentation before attempting for any given key-value pair. If a parameter is optional, the entire key-value can be omitted from the JSON file and the default value will be used.

For calculated values, the user can either keep the key with a dummy value or remove the key entirely when setting up JSON files for a pipeline run.

JSON Object names, keys, and values (e.g., filename strings) are case-sensitive.

The order of key-value pairs within a JSON Object does not matter. Saving/loading the file to/from memory is likely to reorder the contents of the file.

3.2.1 run.json

Named file: `config/user/runs/<run_index>.json`

Purpose

Instructs the pipeline on which input data and user-defined parameters to use in a single program "run", where one "run" configuration serves a single ***Sample*** and a list of ***Model(s)*** and ***Sims(s)***. Enables operational control (breakpoints, which FEM files to save/discard). Keeps track of successful/failed FEMs in Java.

Syntax

To declare this entity in `config/user/runs/`, use the following syntax:

```
{
  "pseudonym": String,
  "submission_context": String,
  "sample": Integer, // note, only one value here!
  "models": [Integer, ...], // pipeline will create all pairwise combos of ...
  "sims": [Integer, ...], // ... models and sims
  "recycle_meshes": Boolean,
  "break_points": {
```

(continues on next page)

(continued from previous page)

```
    "pre_java": Boolean, // before Runner’s handoff() method to Java/COMSOL
    "pre_geom_run": Boolean, // immediately before geometry operations
    "post_geom_run": Boolean, // immediately after geometry operations
    "pre_mesh_proximal": Boolean, // immediately before mesh prox operations
    "post_mesh_proximal": Boolean, // immediately post mesh prox operations
    "pre_mesh_distal": Boolean, // immediately before mesh dist operations
    "post_mesh_distal": Boolean, // immediately post mesh dist operations
    "post_material_assign": Boolean, // immediately post assigning materials
    "pre_loop_currents": Boolean // immediately before solving for bases
  },
  "models_exit_status": [Boolean, ...], // one entry for each Model
  "endo_only_solution": Boolean,
  "keep": {
    "debug_geom": Boolean,
    "mesh": Boolean,
    "bases": Boolean
  },
  "export_behavior": String,
  "partial_fem": {
    "cuff_only": Boolean,
    "nerve_only": Boolean
  },
  "local_avail_cpus": Integer,
  "popup_plots": Boolean,
  "auto_submit_fibers": Boolean
}
```

Properties

`"pseudonym"`: This value (String) informs pipeline print statements, allowing users to better keep track of the purpose of each configuration file. Optional.

`"submission_context"`: The value (String) of this property tells the system how to submit the n_sim NEURON jobs based on the computational resources available. Value can be “cluster”, “local”, or “auto” (if “auto”, “hostname_prefix” is required). Required.

`"hostname_prefix"`: This value (String) tells the program what prefix to look out for in the “HOSTNAME” environment variable. If the “HOSTNAME” begins with the “hostname prefix”, submission context is set to cluster, otherwise it is set to local (does not change the value in the configuration file). Example: if your high performance computing cluster hostname always begins with ourclust, e.g. ourclust-node-15, ourclust-login-20, etc., you would set the value of this variable to “ourclust.” If the “HOSTNAME” environment variable is not present, the program defaults to “local.” Required if “submission_context” is “auto”, otherwise Optional.

`"sample"`: The value (Integer) of this property sets the sample configuration index (”***Sample***”). Note that this is only ever one value. To loop ***Samples***, create a ***Run*** for each. Required.

`"models"`: The value ([Integer, . . .]) of this property sets the model configuration indices (”***Model***”). Required.

`"sims"`: The value ([Integer, . . .]) of this property sets the simulation configuration indices (”***Sim***”). Required.

`"recycle_meshes"`: The value (Boolean) of this property instructs the pipeline to search for mesh matches for recycling a previously generated FEM mesh if set to true. If this property is not specified, the default behavior of the pipeline is false, meaning that it will not search for and recycle a mesh match (see `ModelSearcher` (*Java Utility Classes*) and `mesh_dependent_model.json` (*Mesh Dependent Model*)). Optional.

`"break_points"`: The value (Boolean) of each breakpoint results in the program terminating or continuing with the next ***Model*** index. In Runner, the program checks that at most one breakpoint is true and throws an exception otherwise. The breakpoint locations enable the user to run only up to certain steps of the pipeline, which can be particularly useful for debugging. If a breakpoint is not defined, the default behavior is false (i.e., the pipeline continues beyond the breakpoint). Note: specifying a break point via command line arguments will override any break points set in your run config. Optional.

`"endo_only_solution"`: The value (Boolean) determines what data the electric currents solution will save. Since fibers are sampled from the endoneurium, after the solution is completed, only the endoneurial Ve data is necessary to run fiber simulations. If `"endo_only_solution"` is `true`, then COMSOL will save ONLY the Ve data for the endoneurium. If `false`, COMSOL will save Ve data for the entire model. Recommended value is `true` unless you intend to generate plots or other analyses which require Ve data for geometry other than the endoneurium. If this key-value pair is not present, defaults to `false`. Note: this parameter only affects storage space after the solution has completed, and will not have any affect on memory usage or solution time.

`"models_exit_status"`: The value ([Boolean, . . .]) of this property indicates if Java successfully made the FEMs for the corresponding model indices ("models" property). The user does not need to include this property before performing a run of the pipeline, as it is automatically added in Java (COMSOL FEM processes) and is then used to inform Python operations for making NEURON simulations. The value will contain one value for each ***Model*** listed in "models". If a ***Model*** fails, the pipeline will skip it and proceed to the next one. Automatically added.

`"keep"`: The value (Boolean) of each property results in the program keeping or deleting large COMSOL `*.mph` files for the `"debug_geom.mph"`, `"mesh.mph"` and bases/ for a given ***Model***. If a keep property is not defined, the default behavior is true and the associated `*.mph` file is saved. If `"mesh.mph"` is saved, the file can later be used if another ***Model*** is a suitable "mesh match" and `"recycle_meshes"` is true (see `ModelSearcher` (*Java Utility Classes*) and `mesh_dependent_model.json` (*Mesh Dependent Model*)). If bases/ are saved, a new ***Sim*** for a previously computed ***Sample*** and ***Model*** can be probed along new fibersets/ to create potentials/. Optional.

`"export_behavior"`: The value (String) instructs the pipeline how to behave if an export n_sim directory (i.e., ASCENT_NSIM_EXPORT_PATH/n_sims/) already exists. There are three options: `"selective"` is the default behavior, output directories which already exist will be skipped, but any which do not exist will be generated, `"overwrite"` will remove the extant directory, and generate a new clean output directory in its place, `"error"` instructs the pipeline to exit if any export n_sim directory is found to already exist.

`"partial_fem"`: The value (Boolean) of each property results in the program terminating after building the COMSOL FEM geometry for only the cuff (`"cuff_only"`) or only the nerve (`"nerve_only"`). The program terminates after the `"debug_geom.mph"` file is created. If the `"partial_fem"` JSON Object is not included, the value of each Boolean is treated as false, meaning that the `"debug_geom.mph"` file will contain the nerve and cuff electrode. An error is thrown if the user may set both values for `"cuff_only"` and `"nerve_only"` to true. To build the geometry of both the cuff and the nerve, but not proceed with meshing or solving the FEM, the user should set the value for `"post_geom_run"` under `"break_points"` to true. Overriden if the `"partial_fem"` command line argument is used. Optional.

`"local_avail_cpus"`: The value (Integer) sets the number of CPUs that the program will take if the `"submission_context"` is "local". We check that the user is not asking for more than one less that the number of CPUs of their machine, such that at least one CPU is left for the machine to perform other processes. Optional, but if using submitting locally, the program will take all CPUs except 1 if this value is not defined.

`“popup_plots”`: The value (Boolean) will instruct the pipeline to display plots (e.g. sample plot, fiberset plot, waveform plot) in a popup window. This is in addition to saving the plots in the relevant folders (i.e., the sample and sim folders).

`"auto_submit_fibers"`: The value (Boolean), if true, will cause the program to automatically start fiber simulations after each run is completed. If submitting locally, the program will not continue to the next run until all fiber simulations are complete. If submitting via a computer cluster, the next run will start after all batch NEURON jobs are submitted.

Example

```
{
  "pseudonym":"template_run",
  "submission_context": "local",
  "sample": 62,
  "models": [0],
  "sims": [99],
  "recycle_meshes": true,
  "break_points": {
    "pre_java": false,
    "pre_geom_run": false,
    "post_geom_run": false,
    "pre_mesh_proximal": false,
    "post_mesh_proximal": false,
    "pre_mesh_distal": false,
    "post_mesh_distal": false,
    "post_material_assign": false,
    "pre_loop_currents": false
  },
  "models_exit_status": [true],
  "endo_only_solution":true,
  "keep": {
    "debug_geom": true,
    "mesh": true,
    "bases": true
  },
  "partial_fem": {
    "cuff_only": false,
    "nerve_only": false
  },
  "local_avail_cpus": 3,
  "popup_plots": false,
  "export_behavior": "selective"
}
```

3.2.2 sample.json

Named file: `samples/<sample_index>/sample.json`

Purpose

Instructs the pipeline on which sample-specific input data and user-defined parameters to use for processing nerve sample morphology inputs in preparation for 3D representation of the sample in the FEM (***Sample***) (*Creating Nerve Morphology in COMSOL*).

Syntax

To declare this entity in `samples/<sample_index>/sample.json`, use the following syntax:

```
{
  "sample": String,
  "pseudonym": String,
  "sex": String,
  "level": String,
  "scale": {
    "scale_bar_length": Double,
    "scale_ratio": Double,
    "shrinkage": Double
  },
  "boundary_separation": {
    "fascicles": Double,
    "nerve": Double
  },
  "modes": {
    "mask_input": String,
    "scale_input": String,
    "nerve": String,
    "deform": String,
    "ci_perineurium_thickness": String,
    "reshape_nerve": String,
    "shrinkage_definition": String
  },
  "smoothing": {
    "nerve_distance": Double,
    "fascicle_distance": Double
  },
  "image_preprocessing": {
    "fill_holes": Boolean,
    "object_removal_area": Integer
  },
  "morph_count": Integer,
  "deform_ratio": Double,
  "plot": Boolean,
  "plot_folder": Boolean,
  "render_deform": Boolean,
  "Morphology": {
    "Nerve": {
      "Area": Double
    },
    "Fascicles": [
      {
        "outer": {
          "area": Double,
          "x": Double,
          "y": Double,
          "a": Double,
          "b": Double,
          "angle": Double
```

(continues on next page)

(continued from previous page)

```
        },
        "inners": [
          {
            "area": Double,
            "x": Double,
            "y": Double,
            "a": Double,
            "b": Double,
            "angle": Double
          },
          ...
        ]
      },
      ...
    ]
  }
}
```

Properties

“`sample`”: The value (String) of this property sets the sample name/identifier (e.g., “Rat1-1”) to relate to bookkeeping for input morphology files (*Data Hierarchy Figure A*). The value must match the directory name in `input/<NAME>/` that contains the input morphology files. Required.

"`pseudonym`": This value (String) informs pipeline print statements, allowing users to better keep track of the purpose of each configuration file. Optional.

“`sex`”: The value (String) of this property assigns the sex of the sample. Optional, for user records only.

“`level`”: The value (String) of this property assigns the location of the nerve sample (e.g., cervical, abdominal, pudendal). Optional, for user records only.

“`scale`”

- “`scale_bar_length`”: The value (Double, units: micrometer) is the length of the scale bar in the binary image of the scale bar provided (`s.tif`, note that the scale bar must be oriented horizontally). Required if scale_input = “MASK”.
- “`scale_ratio`”: The value (Double, units: micrometers/pixel) is the ratio of micrometers per pixel for the input mask(s). Required if scale_input = “RATIO”.
- “`shrinkage`”: The value (Double) is the shrinkage correction for the nerve morphology binary images provided as a decimal (e.g., 0.20 results in a 20% expansion of the nerve, and 0 results in no shrinkage correction of the nerve). Required, must be greater than 0. Note: Shrinkage correction scaling is linear (i.e. a nerve with diameter d and area a scaled by scaling factor s will have a final diameter of d_final=d*(1+s) and a final area a_final = a*(1+s)2)

“`boundary_separation`”

- “`fascicles`”: The value (Double, units: micrometer) is the minimum distance required between boundaries of adjacent fascicles (post-deformation, see Deformable in *Python Morphology Classes*)). Required for samples with multiple fascicles.
  - Note that this is a distinct parameter from “`min_fascicle_separation`” in `mock_sample.json`, which controls the minimum distance between fascicles in the binary image mask inputs to the pipeline, which is later deformed to fit in the cuff electrode using Deformable (*Python Morphology Classes*)).

- `“nerve”`: The value (Double, units: micrometer) is the minimum distance required between the boundaries of a fascicle and the nerve (post-deformation, see Deformable in *Python Morphology Classes*)). Required if “nerve” in ***Sample*** is “PRESENT”.

`“modes”`:

- `“mask_input”`: The value (String) is the `“MaskInputMode”` that tells the program which segmented histology images to expect as inputs for fascicles. Required.
  - As listed in Enums (*Enums*), modes include
    1. `“INNERS”`: Program expects segmented images of only inner fascicle boundaries.
    2. `“INNER_AND_OUTER_SEPARATE”`: Program expects segmented image of inners in one file and segmented image of outers in another file.
    3. `“INNER_AND_OUTER_COMPILED”`: Program expects a single segmented image containing boundaries of both inners and outers.
- `“scale_input”`: The value (String) is the `“ScaleInputMode”` that tells the program which type of scale input to look for.
  - As listed in Enums (*Enums*), known `“ScaleInputModes”` include
    1. `“MASK”`: The program will determine image scale from the user’s scale bar mask image and the `scale_bar_length` parameter.
    2. `“RATIO”`: The program will use the scale directly specified in `scale_ratio`. If using this option, a scale bar image need not be provided.
- `“nerve”`: The value (String) is the `“NerveMode”` that tells the program if there is an outer nerve boundary (epineurium) segmented image to incorporate into the model. Required.
  - As listed in Enums (*Enums*), known modes include
    1. `“PRESENT”`: Program expects a segmented image for a nerve (`n.tif`) to incorporate into the model. The value must be PRESENT if multifascicular nerve, but can also be PRESENT if monofascicular.
    2. `“NOT_PRESENT”`: Program does not try to incorporate a nerve boundary into the model. The value cannot be `NOT_PRESENT` if multifascicular nerve, but can be `NOT_PRESENT` if monofascicular.
- `“deform”`: The value (String) is the `“DeformationMode”` that tells the program which method to use to deform the nerve within the cuff. If the `“NerveMode”` (i.e., “nerve” parameter) is defined as `“NOT_PRESENT”` then the `“DeformationMode”` (i.e., “deform” parameter) must be `“NONE”`. Required.
  - As listed in Enums (*Enums*), known modes include
    1. `“NONE”`: The program does not deform the nerve when it is placed in the cuff electrode. In the pipeline’s current implementation, this should be the value for all nerve samples without epineurium (i.e., `“NOT_PRESENT”` for “nerve”).
    2. `“PHYSICS”`: The program uses a physics-based deformation of the nerve to the final inner profile of the nerve cuff, morphing from the original trace towards a circular trace. In the pipeline’s current implementation, this is the only `“DeformationMode”` for deforming compound nerve samples. See `“deform_ratio”` below; if `deform_ratio = 0`, then the original nerve trace is used and if `deform_ratio = 1`, then the nerve trace will be made circular.
- `“ci_perineurium_thickness”`: The value (String) is the `“PerineuriumThicknessMode”` that tells the program which method to use to define perineurium thickness. Required.
  - As listed in Enums (*Enums*), known `“PerineuriumThicknessModes”` include
    1. `“MEASURED”`: The program determines the average thickness of the perineurium using the provided inner and outer boundaries (i.e., `“MaskInputMode”`: `INNER_AND_OUTER_SEPARATE` or

INNER_AND_OUTER_COMPILED). The thickness is determined by finding the half the difference in diameters of the circles with the same areas as the inner and outer traces of the fascicle.

2. Linear relationships between fascicle diameter and perineurium thickness as stored in `config/system/ci_peri_thickness.json`:

   * "GRINBERG_2008"
   * "PIG_VN_INHOUSE_200523"
   * "RAT_VN_INHOUSE_200601"
   * "HUMAN_VN_INHOUSE_200601"

- "reshape_nerve": The value (String) is the "ReshapeNerveMode" that tells the program which final nerve profile (post-deformation) to use when "deform" is set to "PHYSICS". Required.
  - As listed in Enums (*Enums*), known "ReshapeNerveModes" include
    1. "CIRCLE": The program creates a circular nerve boundary with a preserved cross-sectional area (i.e., for multifascicular nerves/nerves that have epineurium).
    2. "NONE": The program does not deform the nerve boundary (i.e., for monofascicular nerves/nerves that do not have epineurium).
- "shrinkage_definition": The value (String) is the "ShrinkageMode" that tells the program how to interpret the "scale"->"shrinkage" parameter, which is provided as a decimal (i.e., 0.2 = 20%). Optional, but assumes the mode "LENGTH_FORWARDS if omitted, since this was the original behavior before this mode was added.
  - As listed in Enums (*Enums*), known "ShrinkageModes" include
    1. "LENGTH_BACKWARDS": The value for "scale"->"shrinkage" refers to how much the length (e.g., radius, diameter, or perimeter) of the nerve cross section was reduced from the fresh tissue to the imaged tissue.
       * Formula: r_post = r_original * (1-shrinkage)
    2. "LENGTH_FORWARDS": The value for "scale"->"shrinkage" refers to how much the length (e.g., radius, diameter, or perimeter) of the nerve cross section increases from the imaged tissue to the fresh tissue.
       * Formula: r_post = r_original / (1+shrinkage)
    3. "AREA_BACKWARDS": The value for "scale"->"shrinkage" refers to how much the area of the nerve cross section was reduced from the fresh tissue to the imaged tissue.
       * Formula: A_post = A_original * (1-shrinkage)
    4. "AREA_FORWARDS": The value for "scale"->"shrinkage" refers to how much the area of the nerve cross section increases from the imaged tissue to the fresh tissue.
       * Formula: A_post = A_original / (1+shrinkage)

"smoothing": Smoothing is applied via a dilating the nerve/fascicle boundary by a specified distance value and then shrinking it by that same value.

- "nerve_distance": Amount (Double) to smooth nerve boundary. Units of micrometers.
- "fascicle_distance": Amount (Double) to smooth fascicle boundaries (inners and outers). Units of micrometers.

"image_preprocessing": Operations applied to the input masks before analysis.

- "fill_holes": The value (Boolean) indicates whether to fill gaps in the binary masks. If true, any areas of black pixels completely enclosed by white pixels will be turned white.

- “object_removal_area”: The value (Integer) indicates the maximum size of islands to remove from the binary masks. Any isolated islands of white pixels with area less than or equal to object_removal_area will be turned black. Units of pixels.

“morph_count”: The value (Integer) can be used to set the number of intermediately deformed nerve traces between the boundary_start and boundary_end in Deformable (*Python Morphology Classes*)) if the “DeformationMode” is “PHYSCIS”. An excessively large number for morph_count will slow down the deformation process, though a number that is too small could result in fascicles “escaping” from the nerve. Optional; if not specified default value is 36.

“deform_ratio”: The value (Double, range: 0 to 1) can be used to deform a nerve if “DeformationMode” is “PHYSICS” to an intermediately deformed boundary_end. For example, if the “ReshapeNerveMode” is “CIRCLE” and deform_ratio = 1, then the boundary_end will be a circle. However, if the “ReshapeNerveMode” is “CIRCLE” and deform_ratio = 0, the boundary_end will be the original nerve trace. A value between 0 and 1 will result in a partially deformed nerve “on the way” to the final circle nerve boundary. Optional, but default value is 1 if not defined explicitly. Note: if deform_ratio = 0, no changes to the nerve boundary will occur, but the physics system will ensure the requirements in "boundary_separation" are met.

“plot”: The value (Boolean) determines whether the program will generate output plots of sample morphology. If true, plots are generated, if false, no plots are generated

“plot_folder”: The value (Boolean) describes plotting behavior (if enabled). If true, plots are generated in the folder samples/<sample_index>/plots, if false, plots will pop up in a window.

"render_deform": The value (Boolean) if true, causes the pipeline to generated a popup window and display a video of sample deformation as it occurs.

“Morphology”: This JSON Object is used to store information about the area and best-fit ellipse information of the nerve and fascicles (outer and inners). The user does not set values to these JSON structures, but they are a convenient record of the nerve morphometry for assigning model attributes based on the anatomy and data analysis. Units: micrometer2 (area); micrometer (length). User does NOT manually set these values. Automatically populated. The values a and b are the full width and height of the ellipse major and minor axes, respectively (i.e., analogous to diameter rather than radius of a circle).

"rotation": The value (Double) instructs the pipeline to rotate the nerve about its centroid by the specified amount (units = Degrees). This parameter may NOT be used if providing an orientation tif image (See *Morphology Input Files*).

Example

```
{
  "sample": "Rat16-3",
  "pseudonym":"template sample",
  "sex": "M",
  "level": "Cervical",
  "scale": {
    "scale_bar_length": 500,
    "scale_ratio": 12.5,
    "shrinkage": 0
  },
  "boundary_separation": {
    "fascicles": 10.0,
    "nerve": 10.0
  },
  "modes": {
    "mask_input": "INNER_AND_OUTER_COMPILED",
```

(continues on next page)

(continued from previous page)

```
    "scale_input": "MASK",
    "nerve": "NOT_PRESENT",
    "deform": "NONE",
    "ci_perineurium_thickness": "MEASURED",
    "reshape_nerve": "NONE",
    "shrinkage_definition": "LENGTH_BACKWARDS"
  },
  "smoothing": {
    "nerve_distance": 0,
    "fascicle_distance": 0
  },
  "image_preprocessing": {
    "fill_holes": true,
    "object_removal_area": 10
  },
  "morph_count": 36,
  "deform_ratio": 1,
  "render_deform": false,
  "Morphology": null
}
```

3.2.3 model.json

Named file: `samples/<sample_index>/models/<model_index>/model.json`

Purpose

Contains user-defined modes and parameters to use in constructing the FEM (***Model***). We provide parameterized control of model geometry dimensions, cuff electrode, material assignment, spatial discretization of the FEM (mesh), and solution. Additionally, `model.json` stores meshing and solving statistics.

Syntax

To declare this entity in `samples/<sample_index>/models/<model_index>/model.json`, use the following syntax:

```
{
  "pseudonym": String,
  "modes": {
    "rho_perineurium": String,
    "cuff_shift": String,
    "fiber_z": String,
    "use_ci": Boolean
  },
  "medium": {
    "proximal": {
      "distant_ground": Boolean,
      "length": Double,
      "radius": Double
```

(continues on next page)

(continued from previous page)

```
    },
    "distal": {
      "exist": Boolean,
      "distant_ground": Boolean,
      "length": Double,
      "radius": Double,
      "shift": {
        "x": Double,
        "y": Double,
        "z": Double
      }
    }
  },
  "inner_interp_tol": Double,
  "outer_interp_tol": Double,
  "nerve_interp_tol": Double,
  "cuff": {
    "preset": String,
    "rotate": {
      "pos_ang": Double,
      "add_ang": Double
    },
    "shift": {
      "x": Double,
      "y": Double,
      "z": Double
    }
  },
  "min_radius_enclosing_circle": Double,
  "mesh": {
  "quality_measure": String,
  "shape_order": String,
    "proximal": {
      "type": {
        "im": String,
        "name": String
      },
      "hmax": Double,
      "hmin": Double,
      "hgrad": Double,
      "hcurve": Double,
      "hnarrow": Double
    },
    "distal": {
      "type": {
        "im": String,
        "name": String
      },
      "hmax": Double,
      "hmin": Double,
      "hgrad": Double,
      "hcurve": Double,
```

(continues on next page)

(continued from previous page)

```
      "hnarrow": Double
    },
    "stats": {
      "name": String,
      "quality_measure_used": String,
      "number_elements": Double,
      "min_quality": Double,
      "mean_quality": Double,
      "min_volume": Double,
      "volume": Double
      "mesh_times": {
        "distal": Double,
        "proximal": Double
      },

    }
  },
  "frequency": Double,
  "temperature": Double,
  "conductivities": {
    "recess": String,
    "medium": String,
    "fill": String,
    "insulator": String,
    "conductor": String,
    "endoneurium": String,
    "perineurium": {
      "label": String,
      "value": Double
    },
    "epineurium": String
  },
  "solver": {
    "sorder": int,
    "type": String
  },
  "solution": {
    "sol_time": Double,
    "name": String
  }
}
```

Properties

`"pseudonym"`: This value (String) informs pipeline print statements, allowing users to better keep track of the purpose of each configuration file. Optional.

"modes"

- `"rho_perineurium"`: The value (String) is the `"PerineuriumResistivityMode"` that tells the program how to calculate the perineurium conductivity in a frequency and/or temperature dependent way. Required.
  - As listed in Enums (*Enums*), known `"PerineuriumResistivityModes"` include
    - `"RHO_WEERASURIYA"`: Program uses mean of circuits C and D from Weerasuriya 1984 [1] (frog sciatic nerve) (*Perineurium Properties*) to adjust perineurium conductivity to account for temperature and frequency (which are both stored in `model.json`).
    - `"MANUAL"`: Program uses the user-defined value of conductivity in "conductivities" (under "perineurium") with no automated correction for frequency.
- `"cuff_shift"`: The value (String) is the `"CuffShiftMode"` that tells the program how to shift the cuff on the nerve (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*). Required.
  - As listed in Enums (*Enums*), known modes include
    - `"NAIVE_ROTATION_MIN_CIRCLE_BOUNDARY"`: Program shifts the cuff to within a user-defined distance of the minimum bounding circle of the nerve sample. The direction of the shift is defined in the preset cuff JSON file (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*). Since this mode does not align the cuff with the sample centroid, orientation masks (`a.tif`) are ignored.
    - `"NAIVE_ROTATION_TRACE_BOUNDARY"`: Program shifts the cuff to within a user-defined distance of the nerve trace boundary. The direction of the shift is defined in the preset cuff JSON file (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*). Since this mode does not align the cuff with the sample centroid, orientation masks (`a.tif`) are ignored.
    - `"AUTO_ROTATION_MIN_CIRCLE_BOUNDARY"`: Program shifts/rotates the cuff to within a user-defined distance of the minimum bounding circle of the nerve sample to align with the Slide's `"fascicle_centroid"`. The direction of the shift is defined in the preset cuff JSON file (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*).
    - `"AUTO_ROTATION_MIN_TRACE_BOUNDARY"`: Program shifts/rotates the cuff to within a user-defined distance of the nerve trace boundary to align with the Slide's `"fascicle_centroid"`. The direction of the shift is defined in the preset cuff JSON file (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*).
    - `"NONE"`: Program keeps both the nerve centroid and cuff centered at (x,y) =(0,0) and no cuff rotation is performed (*Creating Custom Cuffs* and *Cuff Placement on the Nerve*). Note: This mode will ignore any supplied orientation image (`a.tif`).
- `"fiber_z"`: The value (String) is the `"FiberZMode"` that tells the program how to seed the NEURON fibers along the length of the FEM. In the current implementation of the pipeline, this value must be "EXTRUSION". Required.
- `"use_ci"`: The value (Boolean), if true, tells the program whether to model fascicles (outer) that have a single inner as a sheet resistance (i.e., COMSOL "contact impedance") to simplify the meshing of the model. If the value is false, the program will mesh the perineurium of all fascicles. Note that fascicles that have more than one inner per outer always have a meshed perineurium.

`"medium"`: The medium JSON Object contains information for the size, position, and boundary conditions (i.e., grounded external surface) of the proximal (i.e., cylinder containing the full length of nerve and the cuff electrode) and distal (i.e., surrounding) medium domains. Required.

- `“proximal”`: The proximal JSON Object has keys `“distant_ground”` (Boolean) that sets the outer surface to ground if true (if false, sets the outer surface to “current conservation”, i.e. perfectly insulating), “length” (Double, units: micrometer), and “radius” (Double, units: micrometer) of a cylindrical medium. The proximal cylindrical medium is centered at the origin (x,y,z)=(0,0,0), where the (x,y)-origin is the centroid of the nerve sample (i.e., best-fit ellipse of the Trace/Nerve), and extends into only the positive z-direction for the length of the nerve. A warning is printed to the console if the user sets `“distant_ground”` to true AND a distal domain exists (see “distal” below). Required.
- `“distal”`: The distal JSON Object has keys analogous to the “proximal” JSON Object, but contains additional keys “exist” (Boolean) and “shift” (JSON Object containing key-values pairs (Doubles)). The “exist” key allows the user to toggle the existence of the distal medium, which is intended to be used to apply coarser meshing parameters than in the proximal medium (which contains the nerve and cuff electrode along the full length of the nerve). The “shift” JSON Object provides control for the user to move the distal medium around the proximal medium (x,y,z) (Double, units: micrometer). Optional.

`“inner_interp_tol”`: The value (Double) sets the relative tolerance for the representation of the inner trace(s) in COMSOL. When the value is set to 0, the curve is jagged, and increasing the value of this parameter increases the smoothness of the curve. COMSOL’s “closed curve” setting interpolates the points of the curve with continuous first- and second-order derivatives. Generally, we find an interpolation tolerance in the range of 0.01-0.02 to be appropriate, but the user should check that the interpolation tolerance is set correctly for their input nerve sample morphology. See the COMSOL Documentation for more info. Required.

`“outer_interp_tol”`: The value (Double) sets the relative tolerance for the representation of the outer trace(s) in COMSOL. When the value is set to 0, the curve is jagged, and increasing the value of this parameter increases the smoothness of the curve. COMSOL’s “closed curve” setting interpolates the points of the curve with continuous first- and second-order derivatives. Generally, we find an interpolation tolerance in the range of 0.01-0.02 to be appropriate, but the user should check that the interpolation tolerance is set correctly for their input nerve sample morphology. See the COMSOL Documentation for more info. Required.

`“nerve_interp_tol”`: The value (Double) sets the relative tolerance for the representation of the nerve (i.e. epineurium) trace in COMSOL. When the value is set to 0, the curve is jagged, and increasing the value of this parameter increases the smoothness of the curve. COMSOL’s “closed curve” setting interpolates the points of the curve with continuous first- and second-order derivatives. Generally, we find an interpolation tolerance in the range of 0.001-0.005 to be appropriate, but the user should check that the interpolation tolerance is set correctly for their input nerve sample morphology. See the COMSOL Documentation for more info. Required.

`“cuff”`: The cuff JSON Object contains key-value pairs that define which cuff to model on the nerve in addition to how it is placed on the nerve (i.e., rotation and translation). If the user would like to loop over preset cuff designs, then they must create a ***Model*** (model index) for each cuff preset. Required.

- `“preset”`: The value (String) indicates which cuff to model, selected from the list of filenames of the “preset” cuffs in `config/system/cuffs/<filename>.json` (Fig 3A and *Creating Custom Cuffs*). Required.
- `“rotate”`: Contains two keys: `“pos_ang”` (automatically populated based on “CuffShiftMode”, i.e., “cuff_shift” parameter in ***Model***) and `“add_ang”` (optionally set by user to rotate cuff by an additional angle) (*Cuff Placement on the Nerve*).
  - `“pos_ang”` (Double, units: degrees) is calculated by the pipeline for the “AUTO” CuffShiftModes (*Cuff Placement on the Nerve*).
  - `“add_ang”` (Double, units: degrees) is user-defined and adds additional rotation in the counterclockwise direction. If the parameter is not specified, the default value is 0. Optional.
- `“shift”`: Contains three keys: “x”, “y”, and “z”. Automatically calculated based on “CuffShiftMode”.
  - Each key defines the translation of the cuff in the Cartesian coordinate system (Double, units: micrometer). The values are automatically populated by the pipeline based on the `“CuffShiftMode”` (i.e., `“cuff_shift”` parameter within “modes”). Origin (x,y) = (0,0) corresponds to the centroid of the nerve

sample (or fascicle best-fit ellipse of the Trace if monofascicular), and shift in z-direction is along the proximal medium's (i.e., the nerve's) length relative to the cuff "Center" (defined in preset JSON file).

`"min_radius_enclosing_circle"`: The value (Double, units: micrometer) is automatically defined in the program with the radius of the minimum bounding circle of the sample, which is used for placing the cuff on the nerve for certain CuffShiftModes. Automatically calculated.

`"mesh"`: The mesh JSON Object contains key-value pairs that define COMSOL's meshing parameters (required, user-defined) and resulting meshing statistics (automatically calculated).

- `"quality_measure"`: (String) COMSOL measure to use in calculating mesh quality stats. Options include skewness, maxangle, volcircum, vollength, condition, growth. Default is "vollength" if not specified.
- `"shape_order"`: Order of geometric shape functions (String) (e.g., quadratic). Required.
- `"proximal"`: Meshing parameters for the proximal cylindrical domain (as defined in "medium"). Required ([Assigning Material Properties](../../Running_ASCENT/Info.md#control-of-medium-surrounding-nerve and-cuff-electrode)).
  - `"type"`: JSON Object containing parameters/definitions specific to meshing discretization method (e.g., free tetrahedral "ftet"). We recommend free tetrahedral meshes. Required ([Assigning Material Properties](../../Running_ASCENT/Info.md#control-of-medium-surrounding-nerve and-cuff-electrode)).
    - `"im"`: COMSOL indexing prefix (String) (e.g., free tetrahedral "ftet"). Required.
    - `"name"`: COMSOL system name (String) for the created mesh (e.g., "FreeTet"). Required.
  - `"hmax"`: Maximum element size (Double, units: micrometer). We recommend between 1000-4000. Required.
  - `"hmin"`: Minimum element size (Double, units: micrometer). We recommend 1. Required.
  - `"hgrad"`: Maximum element growth (Double). We recommend between 1.8-2.5. Required.
  - `"hcurve"`: Curvature factor (Double). We recommend 0.2. Required.
  - `"hnarrow"`: Resolution of narrow regions (Double). We recommend 1. Required.
- `"distal"`: Meshing parameters for the distal cylindrical domain (as defined in "medium"). Required if distal domain present (see "medium").
  - `"type"`: JSON Object containing parameters/definitions specific to meshing discretization method (e.g., free tetrahedral "ftet"). We recommend free tetrahedral meshes. Required ([Assigning Material Properties](../../Running_ASCENT/Info.md#control-of-medium-surrounding-nerve and-cuff-electrode)).
    - `"im"`: COMSOL indexing prefix (String) (e.g., free tetrahedral "ftet"). Required.
    - `"name"`: COMSOL system name (String) for the created mesh (e.g., "FreeTet"). Required.
  - `"hmax"`: Maximum element size (Double, units: micrometer). We recommend between 1000-4000. Required.
  - `"hmin"`: Minimum element size (Double, units: micrometer). We recommend 1. Required.
  - `"hgrad"`: Maximum element growth (Double). We recommend between 1.8-2.5. Required.
  - `"hcurve"`: Curvature factor (Double). We recommend 0.2. Required.
  - `"hnarrow"`: Resolution of narrow regions (Double). We recommend 1. Required.
- `"stats"`: Meshing statistics. See COMSOL documentation for more details. Automatically populated.
  - `"name"`: Mesher identity (String) which generated the mesh (i.e., COMSOL version)
  - `"quality_measure_used"`: `"quality measure"` which was used to calculate mesh statistics. (String)
  - `"number_elements"`: (Integer)

  - "`min_quality`": (Double)
  - "`mean_quality`": (Double)
  - "`min_volume`": (Double)
  - "`volume`": (Double)
  - "`mesh_times`": JSON Object containing key value parirs storing the elapsed time (in milliseconds) for each mesh type (Proximal and Distal).

"`frequency`": Defines the frequency value used for frequency-dependent material conductivities (Double, unit Hz) (*Perineurium Properties*). Required only if "`PerineuriumResistivityMode`" is "`RHO_WEERASURIYA`".

"`temperature`": Defines the temperature of the nerve environment, which is used to define temperature-dependent material conductivity and ion channel mechanisms in NEURON (Double, unit: Celsius). Required (user must verify `*.MOD` files will adjust Q10 values for temperature, verified temperature is 37 °C).

"`conductivities`": The conductivities JSON Object contains key-value pairs that assign materials (either previously defined in `materials.json` (String), or explicitly defined in place (JSON Object)) to material functions (i.e., cuff "insulator", contact "conductor", contact "recess", cuff "fill", "endoneurium", "perineurium", "epineurium"). All materials functions that are assigned to domains in the part primitives are required. If a contact primitive contains a selection for recessed domain, the recess material must be assigned even if there is no recessed domain in the preset's parameterized implementation.

- If the material is defined in place, the JSON Object value is of the following structure:
  - "`label`": Communicates type of material assigned to domains (String). The label is used to assign materials to domains (*Assigning Material Properties*). Required.
  - "`value`": Required. Material conductivity for isotropic materials (String, unit: S/m) OR (String, "anisotropic") with additional keys:
    * "`sigma_x`", "`sigma_y`", "`sigma_z`" each with values (String, unit: S/m). Required.

"`solver`": The solver JSON Object contains key-value pairs to control the solver. Required.

- "`sorder`": Order of solution shape functions (int) (e.g., quadratic = 2). Required.
- `type`: (String) Solver to use. Options are "direct" or "iterative". Defaults to iterative if not provided, which uses less RAM but takes longer. Optional.

"`solution`": The solution JSON Object contains key-value pairs to keep record of FEM solver processes. Automatically populated.

- "`sol_time`": (Double) Time (in milliseconds) elapsed in solving electric currents.
- "`name`": Solver identity (String) used to solve electric currents (i.e., COMSOL version).

Example

Example

```
{
  "pseudonym":"template model",
  "modes": {
    "rho_perineurium": "RHO_WEERASURIYA",
    "cuff_shift": "AUTO_ROTATION_MIN_CIRCLE_BOUNDARY",
    "fiber_z": "EXTRUSION",
    "use_ci": true
  },
```

(continues on next page)

(continued from previous page)

```
  "medium": {
    "proximal": {
      "distant_ground": false,
      "length": 12500,
      "radius": 3000
    },
    "distal": {
      "exist": true,
      "distant_ground": true,
      "length": 12500,
      "radius": 5000,
      "shift": {
        "x": 0,
        "y": 0,
        "z": 0
      }
    }
  },
  "inner_interp_tol": 0.02,
  "outer_interp_tol": 0.02,
  "nerve_interp_tol": 0.002,
  "cuff": {
    "preset": "Purdue.json",
    "rotate": {
      "add_ang": 0
    },
    "shift": {
      "x": 0,
      "y": 0,
      "z": 0
    }
  },
  "mesh": {
    "quality_measure":"vollength",
    "shape_order": "quadratic",
    "proximal": {
      "type": {
        "im": "ftet",
        "name": "FreeTet"
      },
      "hmax": 1600,
      "hmin": 10,
      "hgrad": 2.2,
      "hcurve": 0.2,
      "hnarrow": 1
    },
    "distal": {
      "type": {
        "im": "ftet",
        "name": "FreeTet"
      },
      "hmax": 1600,
```

(continues on next page)

(continued from previous page)

```
      "hmin": 10,
      "hgrad": 2.5,
      "hcurve": 0.2,
      "hnarrow": 1
    }
  },
  "frequency": 1,
  "temperature": 37,
  "conductivities": {
    "recess": "saline",
    "medium": "muscle",
    "fill": "saline",
    "insulator": "silicone",
    "conductor": "platinum",
    "endoneurium": "endoneurium",
    "perineurium": {
      "label": "RHO_WEERASURIYA @ 1 Hz",
      "value": "0.0008703220191470844"
    },
    "epineurium": "epineurium"
  },
  "solver": {
    "sorder": 2,
    "type": "iterative"
  }
}
```

3.2.4 sim.json

Named file: `config/user/sims/<sim_index>.json`

Purpose

Instructs the pipeline on which user-defined parameters to use in constructing NEURON simulation directories (***Sim***). We provide parameterized control of cuff electrode contact weighting, fiber model and placement in the FEM, extracellular stimulation waveform, intracellular stimulation, flags to indicate which outputs to save (e.g., state variables of channel gating mechanisms, transmembrane potential, intracellular stimulation), and stimulation threshold-finding protocol (*Simulation Protocols*).

Syntax

To declare this entity in `config/user/sims/`, use the following syntax:

```
{
  "pseudonym": String,
  "n_dimensions": Integer,
  "active_srcs": {
    "CorTec300.json": [[Double, Double]], // for example
    "default": [[1, -1]]
```

(continues on next page)

(continued from previous page)

```
  },
  "fibers": {
    "mode": String,
    "xy_trace_buffer": Double,
    "z_parameters": {

      // EXAMPLE diameter parameter for defining fixed diameter fibers
      "diameter": [Double],

      // EXAMPLE diameter parameter for TRUNCHNORM (i.e., diameters from a truncated␣
↪normal
      // distribution) which is only compatible for "MRG_INTERPOLATION" myelinated or
      // unmyelinated fiber types)
      "diameter":{
        "mode": "TRUNCHNORM",
        "mu": Double,
        "std": Double,
        "n_std_limit": Double,
        "seed": Integer
      },

      // EXAMPLE diameter parameter for UNIFORM (i.e., diameters from a uniform
      // distribution) which is only compatible for "MRG_INTERPOLATION" myelinated or
      // unmyelinated fiber types)
      "diameter":{
        "mode": "UNIFORM",
        "upper": Double,
        "lower": Double,
        "seed": Integer
      },

      "min": Double,
      "max": Double,
      "full_nerve_length": Boolean,
      "offset": Double, // or "random" for random jitter within +/- 0.5 internodal length
      "absolute_offset": Double,
      "seed": Integer
    },

    // EXAMPLE XY Parameters for CENTROID (from best-fit ellipse of the Trace)
    "xy_parameters": {
      "mode": "CENTROID"
    },

    // EXAMPLE XY Parameters for UNIFORM_COUNT
    "xy_parameters": {
      "mode": "UNIFORM_COUNT",
      "count": Integer,
      "seed": Integer
    },

    // EXAMPLE XY Parameters for WHEEL
```

(continues on next page)

(continued from previous page)

```
    "xy_parameters": {
      "mode": "WHEEL",
      "spoke_count": Integer,
      "point_count_per_spoke": Integer,
      "find_centroid": Boolean, // centroid of inner polygon
      "angle_offset": Double,
      "angle_offset_is_in_degrees": Boolean
    },

    // EXAMPLE XY Parameters for EXPLICIT
    "xy_parameters": {
      "mode": "EXPLICIT",
      "explicit_fiberset_index" : Integer
    },

    // EXAMPLE XY Parameters for UNIFORM_DENSITY
    "xy_parameters": {
      "mode": "UNIFORM_DENSITY",
      "top_down": Boolean,
      // top_down is true
      "target_density": Double,
      "minimum_number": Integer,
      // top_down is false
      "target_number": Integer,
      "maximum_number": Integer,
      // for both top_down is true and false
      "seed": Integer
    }
  },
  "waveform": {
    "global": {
      "dt": Double,
      "on": Double,
      "off": Double,
      "stop": Double
    },

    // EXAMPLE WAVEFORM for MONOPHASIC_PULSE_TRAIN
    "MONOPHASIC_PULSE_TRAIN": {
      "pulse_width": Double,
      "pulse_repetition_freq": Double,
      "digits": Integer
    },

    // EXAMPLE WAVEFORM for SINUSOID
    "SINUSOID": {
      "pulse_repetition_freq": Double,
      "digits": Integer
    },

    // EXAMPLE WAVEFORM for BIPHASIC_FULL_DUTY
    "BIPHASIC_FULL_DUTY": {
```

(continues on next page)

(continued from previous page)

```
      "pulse_repetition_freq": Double,
      "digits": Integer
    },

    // EXAMPLE WAVEFORM for BIPHASIC_PULSE_TRAIN
    "BIPHASIC_PULSE_TRAIN": {
      "pulse_width": Double,
      "inter_phase": Double,
      "pulse_repetition_freq": Double,
      "digits": Integer
    },

    // EXAMPLE WAVEFORM for
    BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW
    "BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW": {
      "pulse_width_1": Double,
      "pulse_width_2": Double,
      "inter_phase": Double,
      "pulse_repetition_freq": Double,
      "digits": Integer
    },

    // EXAMPLE WAVEFORM for EXPLICIT
    "EXPLICIT": {
      "index": Integer,
      "dt_atol": Double,
      "period_repeats ": Integer
    }
  },
  "intracellular_stim": {
    "times": {
      "pw": Double,
      "IntraStim_PulseTrain_delay": Double,
      "IntraStim_PulseTrain_dur": Double
    },
    "pulse_repetition_freq": Double,
    "amp": Double,
    "ind": Integer
  },
  "saving": {
    "space": {
      "vm": Boolean,
      "gating": Boolean,
      "times": [Double],
    },
    "time": {
      "vm": Boolean,
      "gating": Boolean,
      "istim": Boolean,
      "locs": [Double] OR String
    },
    "end_ap_times": {
```

(continues on next page)

(continued from previous page)

```
      "loc_min": Double,
      "loc_max": Double,
      "threshold": Double
    }
    "runtimes": Boolean
  },

  // EXAMPLE PROTOCOL for FINITE_AMPLITUDES
  "protocol": {
    "mode": "FINITE_AMPLITUDES",
    "initSS": Double,
    "dt_initSS": Double,
    "amplitudes": [Double, Double, ...]
  },

  // EXAMPLE PROTOCOL for ACTIVATION\_THRESHOLD
  "protocol": {
    "mode": "ACTIVATION_THRESHOLD", //String
    "initSS": Double,
    "dt_initSS": Double,
    "threshold": {
      "value": Double,
      "n_min_aps": Integer,
      "ap_detect_location": Double
    },
    "bounds_search": {
      "mode": String,
      "top": Double,
      "bottom": Double,
      "step": Double
    },
    "termination_criteria": {
      "mode": "ABSOLUTE_DIFFERENCE",
      "percent": Double
    }
  },

  // EXAMPLE PROTOCOL for BLOCK_THRESHOLD
  "protocol": {
    "mode": "BLOCK_THRESHOLD", // String
    "initSS": Double,
    "dt_initSS": Double,
    "threshold": {
      "value": Double,
      "n_min_aps": Integer,
      "ap_detect_location": Double
    },
    "bounds_search": {
      "mode": String,
      "top": Double,
      "bottom": Double,
      "step": Double
```

(continues on next page)

(continued from previous page)

```
    },
    "termination_criteria": {
      "mode": String,
      "percent": Double
    }
  },
  "supersampled_bases": {
    "generate": Boolean,
    "use": Boolean,
    "dz": Double,
    "source_sim": Integer
  }
}
```

Properties

"pseudonym": This value (String) informs pipeline print statements, allowing users to better keep track of the purpose of each configuration file. Optional.

"n_dimensions": The value (Integer) is the number of parameters in ***Sim*** for which a list is provided rather than a single value. The user sets the number of parameters they are looping over (e.g., if looping over waveform pulse width and fiber diameter, n_dimensions = 2). We included this control to prevent the user from accidentally creating unintended NEURON simulations. The pipeline will only loop over the first n-dimensions. Required.

"active_srcs": The value is a JSON Object containing key-value pairs of contact weightings for preset cuffs. Each value (List[List[Double]]) is the contact weighting used to make extracellular potentials inputs to NEURON simulations. The order of weights matches the order of parts containing point current sources. The values should not exceed +/-1 in magnitude, otherwise an error is thrown. For monopolar cuff electrodes, the value should be either +1 or -1. For cuff electrodes with more than one contact (2+), the sum of weightings should be +1, -1, or 0. If the preset cuff is not a key in active_srcs, the list of contact weightings for the "default" key is used. Required. The potentials/ for a single fiber are calculated in the following way for the default weighting:

"default": [[1, -1]] // [[weight1 (for src 1 on), weight2 (for src 2 on)]]

The value of potentials/ is applied to a model fiber in NEURON multiplied by the stimulation amplitude, which is either from a list of finite amplitudes or a binary search for thresholds (*Simulation Protocols*)

"fibers": The value is a JSON Object containing key-value pairs that define how potentials are sampled in the FEM for application as extracellular potentials in NEURON (i.e., the Cartesian coordinates of the midpoint for each compartment (i.e., section or segment) along the length of the fiber). Required.

- "mode": The value (String) is the "FiberGeometry" mode that tells the program which fiber geometries to simulate in NEURON (*NEURON Fiber Models*). Required.
  - As listed in *Enums*), known modes include
    - "MRG_DISCRETE" (published MRG fiber model)
    - "MRG_INTERPOLATION" (interpolates the discrete diameters from published MRG fiber models)
    - "TIGERHOLM" (published C-fiber model)

        * "SUNDT" (published C-fiber model)
        * "RATTAY" (published C-fiber model)

    - For a user to simulate a novel fiber type using the pipeline, they must define the spatial discretization of points within the `config/system/fiber_z.json` file to match the dimensions of the fiber compartments connected in NEURON. The "FiberGeometry" mode and parameters in the "fibers" JSON Object in ***Sim*** must match the keys in `config/system/fiber_z.json` to select and define a fiber type (e.g., MRG requires specification of fiber "diameters").

- `"xy_trace_buffer"`: The value (Double, units: micrometer) indicates the minimum required distance between the (x,y)-coordinates of a given fiber and the inner's boundary. Since the domain boundaries are modeled in COMSOL as an interpolation curve, the exact morphology boundary coordinates read into COMSOL will be very close to (but not exactly equal to) those used in Python to seed fiber locations. To protect against instances of fibers falling within the nerve boundary traces in the exact Python traces, but not in COMSOL's interpolation of those traces, we provided this parameter. Required.

- `"z_parameters"`: The value is a JSON Object containing key-value pairs to instruct the system in seeding fibers along the length of the nerve. Required.

    - `"diameter"` The value can take multiple forms to define the fiber diameter that the user is simulating in NEURON (*NEURON Fiber Models*). The value can control simulation of either fixed diameter fibers or fibers chosen from a distribution of diameters (note simulating a distribution of fiber diameters is only compatible with `"MRG_INTERPOLATION"`myelinated or unmyelinated fiber types, not `"MRG_DISCRETE"`). In ***Sim***, only one mode of defining fiber diameters can be used. Required.

        * Fixed diameter: the value (Double or List[Double], units: micrometer) is the diameter of the fiber models. If using with `"MRG_DISCRETE"`, the diameters must be members of the set of published diameters.

        * Distribution of diameters: the value is a dictionary of key-value pairs to define the distribution of diameters based on the `"mode"` parameter, which can be either `"TRUNCNORM"` or `"UNIFORM"`.

            · `"TRUNCNORM"`

            · `"mode"`: "TRUNCNORM" (String). Required.

            · `"mu"`: The value (Double, units micrometer) is the mean diameter of the truncated normal distribution. Required.

            · `"std"`: The value (Double, units micrometer) is the diameter standard deviation of the truncated normal distribution. Required.

            · `"n_std_limit"`: The value (Double) is the number of standard deviations from the mean to bound the truncated normal distribution. Required.

            · `"seed"`: The value (Integer, unitless) seeds the random number generator before sampling fiber diameters.

            · `"UNIFORM"`

            · `"mode"`: `UNIFORM"` (String). Required.

            · `"upper"`: The value (Double, units micrometer) is the upper limit on the distribution of diameters. Required.

            · `"lower"`: The value (Double, units micrometer) is the lower limit on the distribution of diameters. Required.

            · `"seed"`: The value (Integer) seeds the random number generator before sampling fiber diameters.

- “`min`”: the value (Double or List[Double], units: micrometer) is the distal extent of the seeded fiber along the length of the nerve closer to z = 0. Optional: if min and max are not both provided then the fiber length is assumed to be the proximal medium length (see `model.json`).
- “`max`”: The value (Double or List[Double] , units: micrometer) is the distal extent of the seeded fiber along the length of the nerve closer to z = length of the proximal medium (as defined in `model.json`, the length of the nerve). Optional: if min and max are not both provided then the fiber length is assumed to be the proximal medium length by default.
- `full_nerve_length`: (Boolean) Optional. If true, suppresses the warning message associated with using the full length nerve when `"min"` and `"max"` are not defined. Must be false or not defined if `"min"` and `"max"` are defined.
- “`offset`”: The value (Double or String) is the fraction of the node-node length (myelinated fibers) or segment length (unmyelinated fibers) that the center coordinate of the fiber is shifted along the z-axis from the longitudinal center of the proximal medium. If the value is “random”, the offset will be randomly selected between +/- 0.5 section/segment length; to avoid the randomized longitudinal placement, set the offset value to ‘0’ for no offset.
- “`absolute_offset`”: The value (Double) is the distance (micrometers) that the center coordinate of the fiber is shifted along the z-axis from the longitudinal center of the proximal medium. This value is additive with `"offset"`. Note that the shift is with respect to the model center. If a negative value is passed, the fiber will be shifted in the -z direction.
- “`seed`”: The value (Integer) seeds the random number generator before any random offsets are created. Required only if “offset” is not defined, in which case the program will use a random offset.

• “`xy_parameters`”: The value is a JSON Object containing key-value pairs to instruct the system in seeding fiber locations at which to sample potentials inside inners in the nerve cross section (Fig 3B). Include only *one* version of this block in your `sim.json` file. Required.

  “`mode`”: The value (String) is the “`FiberXYMode`” that tells the program how to seed fiber locations inside each inner in the nerve cross section. Required.

• As listed in *Enums*), known modes include

  - “`CENTROID`”: Place one fiber at the centroid (i.e., from the best-fit ellipse of the inner) of each inner.
    - No parameters required.
  - “`UNIFORM_COUNT`”: Randomly place the same number of fibers in each inner, regardless of inner’s size.
    - “`count`”: The value (Integer) is the number of fibers per inner. Required.
    - “`seed`”: The value (Integer) is the seed for the random number generator that is instantiated before the point picking algorithm for fiber (x,y)-coordinates starts. Required.
  - “`WHEEL`”: Place fibers in each inner following a pattern of radial spokes out from the geometric centroid.
    - “`spoke_count`”: The value (Integer) is the number of radial spokes. Required.
    - “`point_count_per_spoke`”: The value (Integer) is the number of fibers to place on each spoke out from the centroid (i.e., excluding the centroid). Required.
    - “`find_centroid`”: The value (Boolean), if true, tells the program to include the geometric centroid as a fiber coordinate. Required.
    - “`angle_offset`”: The value (Double, either degrees or radians depending on next parameter, “`angle_offset_is_in_degrees`”) sets the direction of the first spoke in the wheel. The rest of the spokes are equally spaced (radially) based on the “spoke_count”. Required.

      * "angle_offset_is_in_degrees": The value (Boolean), if true, tells the program to interpret "angle_offset" as degrees. If false, the program interprets "angle_offset" in radians. Required.

  - "EXPLICIT": The mode looks for a "<explicit_index>.txt" file in the user created directory (samples/<sample_index>/explicit_fibersets) for user-specified fiber (x,y)-coordinates (see config/templates/explicit.txt). Note, this file is only required if the user is using the "EXPLICIT" "FiberXYMode". An error is thrown if any explicitly defined coordinates are not inside any inners.

      * "explicit_fiberset_index": The value (Integer) indicates which explicit index file to use.

  - "UNIFORM_DENSITY": Place fibers randomly in each inner to achieve a consistent number of fibers per unit area.

      * "top_down": The value (Boolean) dictates how the pipeline determines how many fibers to seed in each inner. Required.

          · If "top_down" is true, the program determines the number of fibers per inner with the "target_density". If the number of fibers in an inner is less than the "minimum_number", then the minimum number is used.

          · "target_density": The value (Double) is the density (fibers/μm2). Required only if "top_down" is true.

          · "minimum_number": The value (Integer) is the minimum number of fibers that the program will place in an inner. Required only if "top_down" is true.

          · If "top_down" is false (i.e., bottom-up), the program determines the number of fibers per unit area (i.e., fiber density) with "target_number" and the area of the smallest inner. If the number of fibers in an inner is more than the "maximum_number", then the maximum number is used.

          · "target_number": The value (Integer) is the number of fibers placed in the smallest inner. Required only if "top_down" is false.

          · "maximum_number": The value (Integer) is the maximum number of fibers placed in the largest inner. Required only if "top_down" is false.

      * "seed": The value (Integer) is the seed for the random number generator that is instantiated before the point picking algorithm for fiber (x,y)-coordinates starts. Required.

"waveform": The waveform JSON Object contains key-value pairs to instruct the system in setting global time discretization settings and stimulation waveform parameters (Fig 3C). Required.

- "global": the value (JSON Object) contains key-value pairs that define NEURON time discretization parameters. Required.

  - "dt": The value (Double, units: milliseconds) is the time step used in the NEURON simulation for fiber response to electrical stimulation. Required.

  - "on": The value (Double, units: milliseconds) is the time when the extracellular stimulation is turned on. Required.

  - "off": The value (Double, units: milliseconds) is the time when the extracellular stimulation is turned off. Required.

  - "stop": The value (Double, units: milliseconds) is the time when the simulation stops. Required.

The user must also provide ***one*** of the following JSON Objects containing "WaveformMode"-specific parameters. The user can only loop parameters for one type of waveform in a ***Sim***.

Note: the "digits" parameter for the following "WaveformModes" sets the precision of the unscaled current amplitude. For waveforms that are only ever +/-1 and 0 (e.g., MONOPHASIC_PULSE_TRAIN, BIPHASIC_FULL_DUTY, `BIPHASIC_PULSE_TRAIN`), the user can represent the waveform faithfully with 1 digit of precision. However, for waveforms that assume intermediate values (e.g., `SINUSOID`, `BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW`, `EXPLICIT`) more digits of precision may be required to achieve numerical accuracy. An excessive number of digits of precision will increase computational load and waste storage resources.

Note: if one of the parameter values in the `"WaveformMode"` JSON Object is a list, then there are `n_sim/` folders created for as many waveforms as parameter values in the list. If more than one parameter value is a list, then there are `n_sim/` folders created for each combination of waveform parameters among the lists (i.e., the Cartesian product).

- `"MONOPHASIC_PULSE_TRAIN"`
  - `"pulse_width"`: The value (Double, or List[Double], units: milliseconds) is the duration ("width") of the monophasic rectangular pulse. Required.
  - `"pulse_repetition_freq"`: The value (Double, or List[Double], units: Hz) is the rate at which individual pulses are delivered. Required.
  - `"digits"`: The value (Integer) is the number of digits of precision used in saving the waveform to file. Required.
- `"SINUSOID"`
  - `"pulse_repetition_freq"`: The value (Double, or List[Double], units: Hz) is the frequency of the sinusoid. Required.
  - `"digits"`: The value (Integer) is the number of digits of precision used in saving the waveform to file. Required.
- `"BIPHASIC_FULL_DUTY"`: This waveform consists of biphasic symmetric rectangular pulses, where there is no "off" time between repetitions of the biphasic pulse, hence the "full duty cycle" designation. Thus, the phase width (i.e., the duration of one phase in the symmetric pulse) is equal to half of the period, as defined by the specified `"pulse_repetition_freq"`. This waveform is a special case of `"BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW"` (below).
  - `"pulse_repetition_freq"`: The value (Double, or List[Double], units: Hz) is the rate at which individual pulses are delivered. Required.
  - `"digits"`: The value (Integer) is the number of digits of precision used in saving the waveform to file. Required.
- `"BIPHASIC_PULSE_TRAIN"`: This waveform consists of biphasic symmetric rectangular pulses, where the phase width (i.e., the duration of one phase of the symmetric pulse) is defined by `"pulse_width"` and the two phases of the biphasic symmetric pulses may be spaced by a gap defined by `"inter_phase"`. This waveform is a special case of `"BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW"` (below).
  - `"pulse_width"`: The value (Double, or List[Double], units: milliseconds) is the duration ("width") of one phase in the biphasic rectangular pulse. Required.
  - `"inter_phase"`: The value (Double, or List[Double], units: milliseconds) is the duration of time between the first and second phases in the biphasic rectangular pulse. Required.
  - `"pulse_repetition_freq"`: The value (Double, or List[Double], units: Hz) is the rate at which individual pulses are delivered. Required.
  - `"digits"`: the value (Integer) is the number of digits of precision used in saving the waveform to file. Required.
- `"BIPHASIC_PULSE_TRAIN_Q_BALANCED_UNEVEN_PW"`: This waveform consists of biphasic rectangular pulses that are charged-balanced (i.e., the charge of the first phase is equal to the charge of the second phase), but can be defined to be asymmetrical, such that the two phases can have different durations, as defined by `"pulse_width_1"` and `"pulse_width_2"`. Further, the two phases of the biphasic pulses may be spaced by a gap defined by `"inter_phase"`.

  - "pulse_width_1": The value (Double, or List[Double], units: milliseconds) is the duration ("width") of the first phase (positive amplitude) in the biphasic rectangular pulse. Required.
  - "pulse_width_2": The value (Double, or List[Double], units: milliseconds) is the duration ("width") of the second phase (negative amplitude) in the biphasic rectangular pulse. Required.
  - "inter_phase": The value (Double, or List[Double], units: milliseconds) is the duration of time between the primary and secondary phases in the biphasic rectangular pulse (amplitude is 0). Required.
  - "pulse_repetition_freq": The value (Double, or List[Double], units: Hz) is the rate at which individual pulses are delivered. Required.
  - "digits": The value (Integer) is the number of digits of precision used in saving the waveform to file. Required.
- "EXPLICIT"
  - "index": The value (Integer) is the index of the explicit user-provided waveform stored in `config/user/waveforms/<waveform index>.dat` with the time step on the first line followed by the current amplitude value (unscaled, maximum amplitude +/- 1) at each time step on subsequent lines. Required.
  - "dt_atol": The value (Double, units: milliseconds) is the tolerance allowed between the time step defined in the explicit waveform file and the timestep used in the NEURON simulations (see "global" above). If the difference in time step is larger than "dt_atol", the user's explicit waveform is interpolated and resampled at the "global" timestep used in NEURON using SciPy's Signal Processing package (`scipy.signal`) [2]. Required.
  - "period_repeats": The number of times (Integer) the input signal is repeated between when the stimulation turns "on" and "off". The signal is padded with zeros between simulation start (i.e., t=0) and "on", as well as between "off" and "end". Additionally, the signal is padded with zeros between "on" and "off" to accommodate for any extra time after the number of period repeats and before "off". Required.

"intracellular_stim": The value (JSON Object) contains key-value pairs to define the settings of the monophasic pulse train of the intracellular stimulus (*NEURON Scripts*). Required.

- "times": The key-value pairs define the time durations characteristic of the intracellular stimulation. Required.
  - "pw": The value (Double, units: milliseconds) defines the pulse duration of the intracellular stimulation. Required.
  - "IntraStim_PulseTrain_delay": The value (Double, units: milliseconds) defines the delay from the start of the simulation (i.e., t=0) to the onset of the intracellular stimulation. Required.
  - "IntraStim_PulseTrain_dur": The value (Double, units: milliseconds) defines the duration from the start of the simulation (i.e., t=0) to the end of the intracellular stimulation. Required.
- "pulse_repetition_freq": The value (Double, units: Hz) defines the intracellular stimulation frequency. Required.
- "amp": The value (Double, units: nA) defines the intracellular stimulation amplitude. Required.
- "ind": The value (Integer) defines the section index (unmyelinated) or node of Ranvier number (myelinated) receiving the intracellular stimulation. The number of sections/nodes of Ranvier is indexed from 0 and starts at the end of the fiber closest to z = 0. Required.

"saving": The value (JSON Object) contains key-value pairs to define which state variables NEURON will save during its simulations and at which times/locations (*NEURON Scripts*). Required.

- "space":
  - "vm": The value (Boolean), if true, tells the program to save the transmembrane potential at all segments (unmyelinated) and sections (myelinated) at the time stamps defined in "times" (see below). Required.

  - `"gating"`: The value (Boolean), if true, tells the program to save channel gating parameters at all segments (unmyelinated) and sections (myelinated) at the time values defined in "times" (see below). Note: Only implemented for MRG fibers. Required.
  - `"times"`: The value (List[Double], units: milliseconds) contains the times in the simulation at which to save the values of the state variables (i.e., "gating" or "vm") that the user has selected to save for all segments (unmyelinated) and sections (myelinated). Required.

- `"time"`:
  - `"vm"`: The value (Boolean), if true, tells the program to save the transmembrane potential at each time step at the locations defined in "locs" (see below). Required.
  - `"gating"`: The value (Boolean), if true, tells the program to save the channel gating parameters at each time step at the locations defined in "locs" (see below). Note: Only implemented for MRG fibers. Required.
  - `"istim"`: The value (Boolean), if true, tells the program to save the applied intracellular stimulation at each time step. Required.
  - `"locs"`: The value (List[Double] or String, units: none) contains the locations (defined as a decimal percentage, e.g., 0.1 = 10% along fiber length) at which to save the values of the state variables that the user has selected to save for all timesteps. Alternatively, the user can use the value "all" (String) to prompt the program to save the state variables at all segments (unmyelinated) and sections (myelinated). Required.
- `"end_ap_times"`:
  - `"loc_min"`: The value (Double) tells the program at which location to save times at which Vm passes the threshold voltage (defined below) with a positive slope. The value must be between 0 and 1, and less than the value for `"loc_max"`. Be certain not to record from the end section (i.e., 0) if it is passive. A value 0 corresponds to z=0, and a value of 1 corresponds to z=length of proximal domain. Required if this JSON object (which is optional) is included.
  - `"loc_max"`: The value (Double) tells the program at which location to save times at which Vm passes the threshold voltage (defined below) with a positive slope. The value must be between 0 and 1, and greater than the value for `"loc_min"`. Be certain not to record from the end section (i.e., 1) if it is passive. A value 0 corresponds to z=0, and a value of 1 corresponds to z=length of proximal domain. Required if this JSON object (which is optional) is included.
  - `"threshold"`: The value (Double, units: mV) is the threshold value for Vm to pass for an action potential to be detected. Required if this JSON object (which is optional) is included.
- `"runtimes"`: The value (Boolean), if true, tells the program to save the NEURON runtime for either the finite amplitude or binary search for threshold simulation. If this key-value pair is omitted, the default behavior is False.

`"protocol"`:

- `"mode"`: The value (String) is the `"NeuronRunMode"` that tells the program to run activation thresholds, block thresholds, or a list of extracellular stimulation amplitudes (*NEURON Scripts*). Required.
  - As listed in Enums (*Enums*), known `"NeuronRunModes"` include
    * `"ACTIVATION_THRESHOLDS"`
    * `"BLOCK_THRESHOLDS"`
    * `"FINITE_AMPLITUDES"`
- `"initSS"`: The value (Double, hint: should be negative or zero, units: milliseconds) is the time allowed for the system to reach steady state before starting the simulation proper. Required.

- “`dt_initSS`”: The value (Double, units: milliseconds) is the time step (usually larger than “dt” in “global” JSON Object (see above)) used to reach steady state in the NEURON simulations before starting the simulation proper. Required.
- “`amplitudes`”: The value (List[Double], units: mA) contains extracellular current amplitudes to simulate. Required if running “`FINITE_AMPLITUDES`” for “`NeuronRunMode`”.
- “`threshold`”: The JSON Object contains key-value pairs to define what constitutes threshold being achieved. Required for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” and “`BLOCK_THRESHOLDS`”) only.
  - “`value`”: The value (Double, units: mV) is the transmembrane potential that must be crossed with a rising edge for the NEURON code to count an action potential. Required for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” and “`BLOCK_THRESHOLDS`”) only.
  - “`n_min_aps`”: The value (Integer) is the number of action potentials that must be detected for the amplitude to be considered above threshold. Required for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” and “`BLOCK_THRESHOLDS`”) only.
  - “`ap_detect_location`”: The value (Double) is the location (range 0 to 1, i.e., 0.9 is 90% of the fiber length in the +z-direction) where action potentials are detected for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” or “`BLOCK_THRESHOLDS`”). Note: If using fiber models with passive end nodes, the user should not try to detect action potentials at either end of the fiber. Required for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” and “`BLOCK_THRESHOLDS`”) only.
- “`bounds_search`”: the value (JSON Object) contains key-value pairs to define how to search for upper and lower bounds in binary search algorithms (*Simulation Protocols*). Required for threshold finding protocols (i.e., “`ACTIVATION_THRESHOLDS`” and “`BLOCK_THRESHOLDS`”).
  - “`mode`”: the value (String) is the “`SearchAmplitudeIncrementMode`” that tells the program how to change the initial upper and lower bounds for the binary search; the bounds are adjusted iteratively until the initial upper bound (i.e., “top”) activates/blocks and until the initial lower bound does not activate/block, before starting the binary search (*Simulation Protocols*). Required.
    - As listed in Enums (*Enums*), known “`SearchAmplitudeIncrementModes`” include:
      - “`ABSOLUTE_INCREMENT`”: If the current upper bound does not activate/block, increase the upper bound by a fixed “step” amount (e.g., 0.001 mA). If the lower bound activates/blocks, decrease the lower bound by a fixed “step” amount (e.g., 0.001 mA).
      - “`PERCENT_INCREMENT`”: If the upper bound does not activate/block, increase the upper bound by a “step” percentage (e.g., 10 for 10%). If the lower bound activates/blocks, decrease the lower bound by a “step” percentage (e.g., 10 for 10%).
  - “`top`”: The value (Double) is the upper-bound stimulation amplitude first tested in a binary search for thresholds. Required.
  - “`bottom`”: The value (Double) is the lower-bound stimulation amplitude first tested in a binary search for thresholds. Required.
  - “`step`”: The value (Double) is the incremental increase/decrease of the upper/lower bound in the binary search. Required.
    - If “`ABSOLUTE_INCREMENT`”, the value (Double, unit: mA) is an increment in milliamps.
    - If “`PERCENT_INCREMENT`”, the value (Double, units: %) is a percentage (e.g., 10 is 10%).
  - `"max_steps"`: The value (Integer) is the number iterations that will be used in search of a threshold. If the program does not find search bounds which encapsulate threshold (i.e., one bound activates and the other does not) within this number of iterations, the search protocol will exit.
  - “`scout`”: The value (JSONObject) is a set of key-value pairs specifying a previously run “scout” Sim for the same Sample. Use this feature to reduce CPU time required for many fibers placed in the same

inner for a new Sim or Model. If this key is present, ASCENT will look for thresholds in n_sim folders in your ASCENT_NSIM_EXPORT_PATH (i.e., the path you set in config/system/env.json) for fiber0 of each inner for each n_sim. The program will load this threshold value and use it as a starting point for the threshold bounds for each inner in your new Sim. The upper- and lower-bounds in the binary search will be "bounds_search" -> "step" % higher and lower, respectively, of the scout Sim's fiber0 threshold. All parameters except fiber location (i.e., "fibers" -> "xy_location" in Sim) must match between the scout Sim and the current Sim, since the n_sim indices must match between the scout Sim and current Sim. If this key ("scout") is present, the program will ignore the threshold bounds "top" and "bottom" in "protocol" -> "bounds_search" (Unless the specified scout Sim is not found, in which case "top" and "bottom" will be used). Optional.

    * `"model"`: The value (Integer) indicates which model index to use for the scout Sim. Required if `"scout"` is used.
    * `"sim"`: The value (Integer) indicates which sim index to use fo the scout Sim (can be the current Sim's index). Required if `"scout"` is used.

- `"termination_criteria"`: Required for threshold finding protocols (i.e., `"ACTIVATION_THRESHOLDS"` and `"BLOCK_THRESHOLDS"`) (*Simulation Protocols*).
  - `"mode"`: The value (String) is the `"TerminationCriteriaMode"` that tells the program when the upper and lower bound have converged on a solution of appropriate precision. Required.
    * As listed in Enums (*Enums*), known `"TerminationCriteriaModes"` include:
      · `"ABSOLUTE_DIFFERENCE"`: If the upper bound and lower bound in the binary search are within a fixed "tolerance" amount (e.g., 0.001 mA), the upper bound value is threshold.
      · `"tolerance"`: The value (Double) is the absolute difference between upper and lower bound in the binary search for finding threshold (unit: mA). Required.
      · `"PERCENT_DIFFERENCE"`: If the upper bound and lower bound in the binary search are within a relative "percent" amount (e.g., 1%), the upper bound value is threshold. This mode is generally recommended as the `ABSOLUTE_DIFFERENCE` approach requires adjustment of the "tolerance" to be suitable for different threshold magnitudes.
      · `"percent"`: The value (Double) is the percent difference between upper and lower bound in the binary search for finding threshold (e.g., 1 is 1%). Required.

`"supersampled_bases"`: Optional. Required only for either generating or reusing super-sampled bases. This can be a memory efficient process by eliminating the need for long-term storage of the bases/ COMSOL `*.mph` files. Control of `"supersampled_bases"` belongs in ***Sim*** because the (x,y)-fiber locations in the nerve are determined by ***Sim***. The potentials are sampled densely along the length of the nerve at (x,y)-fiber locations once so that in a future pipeline run different fiber types can be simulated at the same location in the nerve cross section without loading COMSOL files into memory.

- `"generate"`: The value (Boolean) indicates if the program will create super-sampled fiber coordinates and super-sampled bases (i.e., sampled potentials from COMSOL). Required only if generating `ss_bases/`.
- `"use"`: The value (Boolean) if true directs the program to interpolate the super-sampled bases to create the extracellular potential inputs for NEURON. If false, the program will sample along the length of the COMSOL FEM at the coordinates explicitly required by "fibers". Required only if generating `ss_bases/`.
- `"dz"`: The value (Double, units: micrometer) is the spatial sampling of the super-sampled bases. Required only if generating `ss_bases/`.
- `"source_sim"`: The value (Integer) is the ***Sim*** index that contains the super-sampled bases. If the user sets both "generate" and "use" to true, then the user should indicate the index of the current ***Sim*** here. Required only if generating `ss_bases/`.

- "termination_criteria": Required for threshold finding protocols (i.e., "ACTIVATION_THRESHOLDS" and "BLOCK_THRESHOLDS") (*Simulation Protocols*).
  - "mode": The value (String) is the "TerminationCriteriaMode" that tells the program when the upper and lower bound have converged on a solution of appropriate precision. Required.
    - As listed in Enums (*Enums*), known "TerminationCriteriaModes" include:
      - "ABSOLUTE_DIFFERENCE": If the upper bound and lower bound in the binary search are within a fixed "tolerance" amount (e.g., 0.001 mA), the upper bound value is threshold.
      - "tolerance": The value (Double) is the absolute difference between upper and lower bound in the binary search for finding threshold (unit: mA). Required.
      - "PERCENT_DIFFERENCE": If the upper bound and lower bound in the binary search are within a relative "percent" amount (e.g., 1%), the upper bound value is threshold. This mode is generally recommended as the ABSOLUTE_DIFFERENCE approach requires adjustment of the "tolerance" to be suitable for different threshold magnitudes.
      - "percent": The value (Double) is the percent difference between upper and lower bound in the binary search for finding threshold (e.g., 1 is 1%). Required.

"supersampled_bases": Optional. Required only for either generating or reusing super-sampled bases. This can be a memory efficient process by eliminating the need for long-term storage of the bases/ COMSOL *.mph files. Control of "supersampled_bases" belongs in ***Sim*** because the (x,y)-fiber locations in the nerve are determined by ***Sim***. The potentials are sampled densely along the length of the nerve at (x,y)-fiber locations once so that in a future pipeline run different fiber types can be simulated at the same location in the nerve cross section without loading COMSOL files into memory.

- "generate": The value (Boolean) indicates if the program will create super-sampled fiber coordinates and super-sampled bases (i.e., sampled potentials from COMSOL). Required only if generating ss_bases/.
- "use": The value (Boolean) if true directs the program to interpolate the super-sampled bases to create the extracellular potential inputs for NEURON. If false, the program will sample along the length of the COMSOL FEM at the coordinates explicitly required by "fibers". Required only if generating ss_bases/.
- "dz": The value (Double, units: micrometer) is the spatial sampling of the super-sampled bases. Required only if generating ss_bases/.
- "source_sim": The value (Integer) is the ***Sim*** index that contains the super-sampled bases. If the user sets both "generate" and "use" to true, then the user should indicate the index of the current ***Sim*** here. Required only if generating ss_bases/.

Example

```
{
  "pseudonym":"template sim",
  "n_dimensions": 1,
  "active_srcs": {
    "CorTec300.json": [[1, -1]],
    "default": [[1, -1]]
  },
  "fibers": {
    "mode": "MRG_DISCRETE",
    "xy_trace_buffer": 5.0,
    "z_parameters": {
      "diameter": [1, 2, 5.7, 7.3, 8.7, 10],
      "min": 0,
```

(continues on next page)

(continued from previous page)

```
      "max": 12500,
      "offset": 0,
      "seed": 123
    },
    "xy_parameters": {
      "mode": "UNIFORM_DENSITY",
      "top_down": true,
      "minimum_number": 1,
      "target_density": 0.0005,
      "maximum_number": 100,
      "target_number": 50,
      "seed": 123
    }
  },
  "waveform": {
    "global": {
      "dt": 0.001,
      "on": 1,
      "off": 49,
      "stop": 50
    },
    "BIPHASIC_PULSE_TRAIN": {
      "pulse_width": 0.1,
      "inter_phase": 0,
      "pulse_repetition_freq": 1,
      "digits": 1
    }
  },
  "intracellular_stim": {
    "times": {
      "pw": 0,
      "IntraStim_PulseTrain_delay": 0,
      "IntraStim_PulseTrain_dur": 0
    },
    "pulse_repetition_freq": 0,
    "amp": 0,
    "ind": 2
  },
  "saving": {
    "space": {
      "vm": false,
      "gating": false,
      "times": [0]
    },
    "time": {
      "vm": false,
      "gating": false,
      "istim": false,
      "locs": [0]
    },
    "end_ap_times": {
      "loc_min": 0.1,
```

(continues on next page)

(continued from previous page)

```
      "loc_max": 0.9,
      "threshold": -30
    },
    "runtimes": false
  },
  "protocol": {
    "mode": "ACTIVATION_THRESHOLD",
    "initSS": -200,
    "dt_initSS": 10,
    "threshold": {
      "value": -30,
      "n_min_aps": 1,
      "ap_detect_location": 0.9
    },
    "bounds_search": {
      "mode": "PERCENT_INCREMENT",
      "top": -1,
      "bottom": -0.01,
      "step": 10,
      "scout_sim": false
    },
    "termination_criteria": {
      "mode": "PERCENT_DIFFERENCE",
      "percent": 1
    }
  },
  "supersampled_bases": {
    "generate": false,
    "use": false,
    "dz": 1.0,
    "source_sim": 1009
  }
}
```

3.2.5 mock_sample.json

Named file: `config/user/<mock_sample_index>.json`

Purpose

Instructs the pipeline on which user-defined parameters to use for synthesizing nerve sample morphology inputs (i.e., 2D binary images of segmented nerve morphology) for a single sample in preparation for 3D representation of the nerve in the FEM.

Syntax

To declare this entity in config/user/<mock_sample_index>.json, use the following syntax:

Note: Eccentricity (e) is defined as a function of the major (a-) and minor (b-) axes as follows:

```
{
  "global": {
    "NAME": String
  },
  "scalebar_length": Double,
  "nerve": {
    "a": Double,
    "b": Double,
    "rot": Double
  },
  "figure": {
    "fig_margin": Double,
    "fig_dpi": Integer
  },

  // EXAMPLE POPULATE Parameters for EXPLICIT
  "populate": {
    "mode": "EXPLICIT",
    "min_fascicle_separation": 5,
    "Fascicles": [
      {
        "centroid_x": Double,
        "centroid_y": Double,
        "a": Double,
        "b": Double,
        "rot": Double
      },
      ...
    ]
  },

  // EXAMPLE POPULATE Parameters for TRUNCNORM
  "populate": {
    "mode": "TRUNCNORM",
    "mu_fasc_diam": Double,
    "std_fasc_diam": Double,
    "n_std_diam_limit": Double,
    "num_fascicle_attempt": Integer,
    "num_fascicle_placed": Integer,
    "mu_fasc_ecc": Double,
```

(continues on next page)

(continued from previous page)

```
    "std_fasc_ecc": Double,
    "n_std_ecc_limit": Double,
    "max_attempt_iter": Integer,
    "min_fascicle_separation": Double,
    "seed": Integer
  },

  // EXAMPLE POPULATE Parameters for UNIFORM
  "populate": {
    "mode": "UNIFORM",
    "lower_fasc_diam": Double,
    "upper_fasc_diam": Double,
    "num_fascicle_attempt": Integer,
    "num_fascicle_placed": Integer,
    "lower_fasc_ecc": Double,
    "upper_fasc_ecc": Double,
    "max_attempt_iter": Integer,
    "min_fascicle_separation": Double,
    "seed": Integer
  }
}
```

Properties

`"global"`: The global JSON Object contains key-value pairs to document characteristics of the sample being synthesized. Required.

- `"NAME"`: The value (String) of this property sets the sample name/identifier (e.g., Pig1-1) to relate to bookkeeping. The value will match the directory name in input/ containing the synthesized morphology files created by the `mock_morphology_generator.py`. Required.

`"scalebar_length"`: The value (Double, units: micrometer) is the desired length of the scale bar in the generated binary image of the scale bar (`s.tif`). Required.

`"nerve"`: The nerve JSON Object contains key-value pairs for the elliptical nerve's size and rotation. Required.

- `"a"`: Value is the nerve ellipse axis 'a' which is the full width major axis (Double, units: micrometer). Required.
- `"b"`: Value is the nerve ellipse axis 'b' which is the full width minor axis (Double, units: micrometer). Required.
- `"rot_nerve"`: Value is the nerve ellipse axis rotation (Double, units: degrees). Positive angles are counterclockwise and negative are clockwise, relative to orientation with a-axis aligned with +x-direction. Required.

"figure": The figure JSON Object contains key-value pairs for parameters that determine the size and resolution of the generated binary masks. Required.

- `"fig_margin"`: The value (Double, >1 otherwise an error is thrown) sets the x- and y-limits of the binary masks generated. The limits are set relative to the maximum nerve ellipse axis dimension (+/- `fig_margin`*max(`a`, `b`) in both x- and y-directions). Required.
- `"fig_dpi"`: The value (Integer) is the "dots per inch" resolution of the synthesized binary images. Higher resolutions will result in more accurate interpolation curves of inners in COMSOL. We recommend >1000 for this value. Required.

`"populate"`: The populate JSON Object contains key-value pairs for parameters that determine how the nerve contents (i.e., fascicle inners) are populated. Required.

- `"mode"`: The value (String) is the `"PopulateMode"` that tells the program which method to use to populate the nerve. Required.
  - As listed in Enums (*Enums*), known `"PopulateModes"` include
    - `"EXPLICIT"`: Populates the nerve with elliptical inners that are defined explicitly by the user.
      - `"min_fascicle_separation"`: The value (Double, units: micrometer) determines the minimum distance between fascicle boundaries in the binary mask image that the pipeline will allow without throwing an error. This value controls the separation required between fascicles in the binary mask only. There is a separate parameter for forcing a distance (post-deformation) of the nerve between fascicles and between fascicles and the nerve boundary in `sample.json`. Required.
      - `"Fascicles"`: The value `List[JSON Object]` contains a JSON Object for each inner. Required. Within each JSON Object, the following key-value pairs are present:
      - `"centroid_x"`: Value (Double, units: micrometer) is the x-coordinate of the centroid of the best-fit ellipse of the Trace. Required.
      - `"centroid_y"`: Value (Double, units: micrometer) is the y-coordinate of the centroid of the best-fit ellipse of the Trace. Required.
      - `"a"`: Value is the ellipse axis 'a' which is the full width major axis (Double, units: micrometer). Required.
      - `"b"`: Value is the ellipse axis 'b' which is the full width minor axis (Double, units: micrometer). Required.
      - `"rot"`: Value is the ellipse axis rotation (Double, units: degrees). Positive angles are counterclockwise and negative are clockwise, relative to orientation with a-axis aligned with +x-direction. Required.
    - `"TRUNCNORM"`: Places fascicles in the nerve with size and eccentricity randomly based on a truncated normal distribution. Rotation of fascicles is randomly drawn from uniform distribution from 0 to 360 degrees.
      - `"mu_fasc_diam"`: The value (Double, units: micrometer) is the mean fascicle diameter in the distribution. Required.
      - `"std_fasc_diam"`: The value (Double, units: micrometer) is the standard deviation of fascicle diameter in the distribution. Required.
      - `"n_std_diam_limit"`: The value (Double) is the limited number of standard deviations for the truncated normal fascicle diameter distribution. The value does not need to be an integer. Required.
      - `"num_fascicle_attempt"`: The value (Integer) is the number of different fascicles from the distribution that the program will attempt to place. Required.
      - If `"num_fascicle_attempt"` does not equal `"num_fascicle_placed"`, a warning is printed to the console instructing user to reduce the number, size, and/or separation of the fascicles.
      - `"num_fascicle_placed"`: The value (Integer) is the number of successfully placed fascicles in the nerve cross section. Automatically populated.
      - If `"num_fascicle_attempt"` does not equal `"num_fascicle_placed"`, a warning is printed to the console instructing user to reduce the number, size, and/or separation of the fascicles.
      - `"mu_fasc_ecc"`: The value (Double) is the mean fascicle eccentricity in the distribution. Must be <= 1 and > 0. Set to 1 for circles. Required.

- "std_fasc_ecc": The value (Double, units: micrometer) is the standard deviation of fascicle eccentricity in the distribution. Required.
- "n_std_ecc_limit": The value (Double) is the limited number of standard deviations for the truncated normal fascicle eccentricity distribution. Required.
- "max_attempt_iter": The value (Integer) is the number of different random locations within the nerve that the program will attempt to place a fascicle before skipping it (presumably because it cannot possibly fit in the nerve). We recommend using 100+. Required.
- "min_fascicle_separation": The value (Double, units: micrometer) determines the minimum distance between fascicle boundaries in the binary mask image that the pipeline will allow without throwing an error. This value controls the separation required between fascicles in the binary mask only. There is a separate parameter for forcing a distance (post-deformation) of the nerve between fascicles and between fascicles and the nerve boundary in `sample.json`. Required.
- "seed": The value (Integer) initiates the random number generator. Required.

* "UNIFORM": Places fascicles in the nerve with size and eccentricity randomly based on a uniform distribution. Rotation of fascicles is randomly drawn from uniform distribution from 0 to 360 degrees.
  - "lower_fasc_diam": The value (Double, units: micrometer) is the lower limit of the uniform distribution for fascicle diameter. Required.
  - "upper_fasc_diam": The value (Double, units: micrometer) is the upper limit of the uniform distribution for fascicle diameter. Required.
  - "num_fascicle_attempt": The value (Integer) is the number of different fascicles from the distribution that the program will attempt to place. Required.
  - If "num_fascicle_attempt" does not equal "num_fascicle_placed", a warning is printed to the console instructing user to reduce the number, size, and/or separation of the fascicles.
  - "num_fascicle_placed": The value (Integer) is the number of successfully placed fascicles in the nerve cross section. Automatically populated.
  - If "num_fascicle_attempt" does not equal "num_fascicle_placed", a warning is printed to the console instructing user to reduce the number, size, and/or separation of the fascicles.
  - "lower_fasc_ecc": The value (Double) is the lower limit of the uniform distribution for fascicle eccentricity. Must be <= 1 and > 0. Set to 1 for circles. Required.
  - "upper_fasc_ecc": The value (Double) is the upper limit of the uniform distribution for fascicle eccentricity. Must be <= 1 and > 0. Set to 1 for circles. Required.
  - "max_attempt_iter": The value (Integer) is the number of different random locations within the nerve that the program will attempt to place a fascicle before skipping it (presumably because it cannot possibly fit in the nerve). We recommend using 100+. Required.
  - "min_fascicle_separation": The value (Double, units: micrometer) determines the minimum distance between fascicle boundaries in the binary mask image that the pipeline will allow without throwing an error. This value controls the separation required between fascicles in the binary mask only. There is a separate parameter for forcing a distance (post-deformation) of the nerve between fascicles and between fascicles and the nerve boundary in `sample.json`. Required.
  - "seed": The value (Integer) initiates the random number generator. Required.

Example

```
{
  "global": {
    "NAME": "Alien1-1"
  },
  "scalebar_length": 1000,
  "nerve": {
    "a": 3000,
    "b": 2000,
    "rot": 45
  },
  "figure": {
    "fig_margin": 1.2,
    "fig_dpi": 1000
  },
  "populate": {
    "mode": "UNIFORM",
    "lower_fasc_diam": 450,
    "upper_fasc_diam": 500,
    "num_fascicle_attempt": 10,
    "num_fascicle_placed": 0,
    "lower_fasc_ecc": 0.5,
    "upper_fasc_ecc": 0.99,
    "max_attempt_iter": 100,
    "min_fascicle_separation": 10,
    "seed": 123
  }
}
```

3.2.6 query_criteria.json

Named file: `config/user/query_criteria/<query criteria index>.json`

Purpose

Used to guide the Query class's searching algorithm in the `run()` and `_match()` methods. This is used for pulling ***Sample***, ***Model***, or ***Sim*** indices for data analysis. The `query_criteria.json` dictates if a given ***Sample***, ***Model***, or ***Sim*** fit the user's restricted parameter values (*Python Morphology Classes*).

Syntax

```
{
  "partial_matches": Boolean,
  "include_downstream": Boolean,
  "sample": { // can be empty, null, or omitted
  },
  "model": { // can be empty, null, or omitted
  },
  "sim": { // can be empty, null, or omitted
```

(continues on next page)

(continued from previous page)

```
  },
  "indices": {
    "sample": null, Integer, or [Integer, ...],
    "model": null, Integer, or [Integer, ...],
    "sim": null, Integer, or [Integer, ...]
  }
}
```

Properties

`"partial_matches"`: The value (Boolean) indicates whether Query should return configuration indices for ***Sample***, ***Model***, or ***Sim*** that are a partial match (i.e., a subset of the parameters were found, but not all).

`"include_downstream"`: The value (Boolean) indicates whether Query should return indices of downstream (***Sample*** > ***Model*** > ***Sim***) configurations that exist if match criteria are not provided for them.

`"sample"`: The value is a JSON Object that mirrors the path to the parameters of interest in ***Sample*** and their value(s).

`"model"`: The value is a JSON Object that mirrors the path to the parameters of interest in ***Model*** and their value(s).

`"sim"`: The value is a JSON Object that mirrors the path to the parameters of interest in ***Sim*** and their value(s).

`"indices"`:

- `"sample"`: The value (null, Integer, or [Integer, . . .]) for explicitly desired ***Sample*** indices
- `"model"`: The value (null, Integer, or [Integer, . . .]) for explicitly desired ***Model*** indices
- `"sim"`: The value (null, Integer, or [Integer, . . .]) for explicitly desired ***Sim*** indices

Note: you can have BOTH lists of desired ***Sample***, ***Model***, and ***Sim*** indices AND search criteria in one `query_criteria.json`.

Example

```
{
  "partial_matches": true,
  "include_downstream": true,
  "sample": {
    "sample": "Rat16-3"
  },
  "model": {
    "medium": {
      "bounds": {
        "radius": [1000, 2000]
      }
    },
    "dummy": 0
  },
  "sim": null,
  "indices": {
    "sample": null,
    "model": null,
    "sim": null
```

(continues on next page)

(continued from previous page)

```
  }
}
```

3.2.7 env.json

Named file: `config/system/env.json`

Purpose

The file contains key-value pairs for paths. The file can be automatically populated by running `env_setup.py` (*Installation*). Note that we have prepended all of the keys in this file with "ASCENT" because these key-value pairs are directly stored as environment variables, so the "ASCENT" key distinguishes these pairs from other paths that may be present on your computer.

Syntax

To declare this entity in `config/system/env.json`, use the following syntax:

```
{
  "ASCENT_COMSOL_PATH": String,
  "ASCENT_JDK_PATH": String,
  "ASCENT_PROJECT_PATH": String,
  "ASCENT_NSIM_EXPORT_PATH": String
}
```

Properties

`"ASCENT_COMSOL_PATH"`: The value (String) is the path for your local COMSOL installation. `"ASCENT_JDK_PATH"`: The value (String) is the path for your local Java JDK installation. `"ASCENT_PROJECT_PATH"`: The value (String) is the path for your local ASCENT repository. `"ASCENT_NSIM_EXPORT_PATH"`: The value (String) is the path where the pipeline will save NEURON simulation directories to submit.

Example

Windows:

```
{
  "ASCENT_COMSOL_PATH": "C:\\Program Files\\COMSOL\\COMSOL55\\Multiphysics",
  "ASCENT_JDK_PATH": "C:\\Program Files\\Java\\jdk1.8.0_221\\bin",
  "ASCENT_PROJECT_PATH": "D:\\Documents\\ascent",
  "ASCENT_NSIM_EXPORT_PATH": "D:\\Documents\\ascent\\submit"
}
```

MacOS

```
{
  "ASCENT_COMSOL_PATH": "/Applications/COMSOL55/Multiphysics ",
  "ASCENT_JDK_PATH": "/Library/Java/JavaVirtualMachines/jdk1.8.0_221.jdk/Contents/Home/
↪bin/",
```

(continues on next page)

(continued from previous page)

```
  "ASCENT_PROJECT_PATH": "/Users/ericmusselman/Documents/ascent",
  "ASCENT_NSIM_EXPORT_PATH": "/Users/ericmusselman/Documents/ascent/submit"
}
```

3.2.8 exceptions.json

Named file: `config/system/exceptions.json`

Purpose

The file contains a list of JSON Objects, one for each documented pipeline exception. Note that this file is a single large Array, so it is wrapped in square brackets, as opposed to all other JSON files, which are wrapped in curly braces.

Syntax

To declare this entity in `config/system/env.json`, use the following syntax:

```
[
  {
    "code": Integer,
    "text": String
  },
  ...
]
```

Properties

`"code"`: The value (Integer) is an identifier that enables easy reference to a specific exception using Exceptionable's `self.throw(<code_index>)`. This value must be unique because the Exeptionable class uses this value to find any given exception. We suggest that you increment the code with each successive exception (akin to indexing), but any number will work as long as its value is unique.

`"text"`: The value (String) is a message to the user explaining why the pipeline failed.

Example

See: `config/system/exceptions.json`

Note: The user should not need to change this file unless adding new exceptions to ASCENT.

3.2.9 materials.json

Named file: `config/system/materials.json`

Purpose

Stores default material and tissue properties in the pipeline for use in the 3D FEM.

Syntax

To declare this entity in `config/system/materials.json`, use the following syntax:

```
{
  "conductivities": {
    "endoneurium": { // example syntax for anisotropic medium
      "value": "anisotropic",
      "sigma_x": String,
      "sigma_y": String,
      "sigma_z": String,
      "unit": "String",
      "references": {
        "1": String,
        ...,
        "n": String
      }
    },
    "epineurium": { // example syntax for isotropic medium
      "value": "String",
      "unit": "String",
      "references": {
        "1": String,
        ...,
        "n": String
      }
    }
  }
}
```

Properties

`"<material>"`: The value is a JSON Object containing the conductivity value and units as Strings. Though using strings may seem odd for storing conductivity values, we do this because we read them directly into COMSOL to define materials, and COMSOL expects a string (which it evaluates as expression).

- `"value"`: The conductivity of the material (if not "anisotropic", the value must be in units S/m). If the value is "anisotropic", the system is expecting additional keys for the values in each Cartesian direction:
  - `"sigma_x"`: The value (String) is the conductivity in the x-direction (unit: S/m)
  - `"sigma_y"`: The value (String) is the conductivity in the y-direction (unit: S/m)
  - `"sigma\_z"`: The value (String) is the conductivity in the z-direction (unit: S/m)
- `"unit"`: The unit of the associated conductivity in square brackets (must be "[S/m]")

- `"references"`: The value (Dictionary) contains citations to the source of the material conductivity used. The contents are non-functional (i.e., they are not used in any of the code), but they serve as a point of information reference for good bookkeeping. Each reference used is assigned its own key-value pair (Optional).

Example

See: `config/system/materials.json` to see all built-in material definitions, which the user may add to.

Note: Perineurium can be represented in the pipeline as either a meshed domain with a finite thickness or as a thin layer approximation, but the conductivity value used for either method is defined in `materials.json` unless the `"PerineuriumResistivityMode"` is `"MANUAL"` and the conductivity is defined explicitly in ***Model*** (*Perineurium Properties*).

3.2.10 ci_peri_thickness.json

Named file: `config/system/ci_peri_thickness.json`

Purpose

The file stores `"PerineuriumThicknessMode"` definitions that are referenced in `sample.json` (`"ci_perineurium_thickness"`) for assigning perineurium thickness values to fascicles for a mask of inners if `"ci_perineurium_thickness"` is not specified as “MEASURED”. The calculated thickness may be explicitly built in the FEM geometry and meshed (i.e., if `"use_ci"` in ***Model*** is false) or may only be used for calculating the contact impedance if modeling the perineurium with a thin layer approximation (*Creating Nerve Morphology in COMSOL*, and *Perineurium Properties*).

Syntax

```
{
  "ci_perineurium_thickness_parameters": {
    "GRINBERG_2008": {
      "a": Double,
      "b": Double
    },
    ...
  }
}
```

Properties

`"<PerineuriumThicknessMode>"`: JSON Object that contains key-value pairs defining the relationship between fascicle diameter (micrometers) and perineurium thickness. Required.

- `"a"`: Value (Double, units: µm/µm) as in thickness = a*x + b
- `"b"`: Value (Double, units: µm) as in thickness = a*x + b

Example

See: `config/system/ci_peri_thickness.json` to see all built-in `PerineuriumThicknessMode` relationships.

3.2.11 mesh_dependent_model.json

Named file: `config/system/mesh_dependent_model.json`

Purpose

This file is not to be changed unless a user adds new parameters to ***Model***. The use of this file happens behind the scenes. The file informs the `ModelSearcher` class (*Java Utility Classes*) if two model configurations constitute a "mesh match" (i.e., that the mesh from a previously solved and identical model can be recycled). Note that if you modify the structure of `model.json`, the pipeline expects this file's structure to be changed as well. If the Boolean for a parameter is true, then the parameter values between two ***Models*** must be identical to constitute a "mesh match". If the Boolean for a parameter is false, then the parameter values between the two ***Models*** can be different and still constitute a "mesh match". The process of identifying "mesh matches" is automated and is only performed if the `"recycle_meshes"` parameter in ***Run*** is true.

Syntax

To declare this entity in `config/system/mesh_dependent_model.json`, use the following syntax:

- The same key-value structure pair as in ***Model***, but the values are of type Boolean
  - true: The parameter values between the two ***Model*** configurations must be identical to constitute a "mesh match".
  - false: The parameter values between the two ***Model*** configurations can be different and still constitute a "mesh match".

Properties

See: `model.json`

Example

See: `config/system/mesh_dependent_model.json`

- Note: The user should not need to change this file unless adding new parameters to ***Model*** for expanded/modified pipeline functionality.

CHAPTER

FOUR

TEMPLATE FOR REPORTING METHODS IN A PUBLICATION

For uses of ASCENT, either stand-alone in its publicly available form, as a starting point upon which further work is developed, or as a component of a system (e.g., connected with other organ models or engineering optimization techniques), we ask that the author(s) cite both the ASCENT publication and the release of ASCENT (e.g., ASCENT v1.0.0). Make sure to use the correct DOI for the release used.

The following guidelines are to help users adhere to FAIR principles when using ASCENT and thereby enable model reproducibility and reduce duplication of efforts [Wilkinson *et al.*, 2016]. With published modeling studies using ASCENT, we ask that users provide all code and input files required to reproduce their work.

To streamline dissemination of work, upon acceptance of a manuscript to a journal, we encourage users to either make their fork/branch of the ASCENT repository publicly available, or to make a "Merge Request" to the ASCENT GitHub repository (i.e., if your developments would be of general interest to ASCENT users) so that we can review and incorporate your changes to our public repository.

4.1 Configuration files

Provide the JSON configuration files used as inputs to ASCENT.

- sample.json
- model.json
- sim.json
- mock_sample.json (if applicable)
- map.json (if serial nerve sections are used to create 3D representation of the nerve in future ASCENT releases)
- ci_peri_thickness.json (if using new relationships between perineurium thickness and inner diameter not already in the ASCENT repository)
- materials.json (if using new materials that have been added to this file and referenced in model.json rather than explicitly defined in model.json)
- fiber_z.json (if using fiber models with ultrastructure that is not integrated in the ASCENT repository)

4.2 Nerve segmentation files

Indicate the software and methods used to generate the binary image inputs (e.g., Adobe Photoshop).

If defining nerve geometry with segmented binary images of nerve microanatomy, indicate the methods for the technique used to obtain the nerve images (i.e., histology, Micro-CT, ultrasound, etc.) and to segment the tissue boundaries.

If referencing a dataset of published nerve microanatomy, cite the original source, include statements to justify the relevance of the source to define nerve tissue boundaries for your model(s), and report high-level statements for the methods used. Indicate the methods used to segment the tissue boundaries.

If using our mock_morphology_generator.py script, please indicate so, and make a statement regarding the methods used to populate the nerve with fascicles. Present either your own data or cite previously published literature used to inform the nerve diameter and the number, size, and placement of fascicle(s) in the nerve.

Explicitly indicate in your methods if you used any correction for tissue shrinkage in preparing nerve inputs for modeling in the FEM. If so, cite literature justifying your assumption. Additionally, indicate any methods used to deform the nerve to fit within the cuff electrode in ASCENT.

Provide the image files used as inputs to ASCENT.

- n.tif (if applicable)
- i.tif or o.tif, both i.tif and o.tif, or c.tif (as applicable)
- s.tif
- a.tif (if applicable)

4.3 Cuff

If the "preset" cuff configuration file used to define the cuff electrode is not publicly available in the ASCENT repository, please provide it with your materials.

If new part primitives were created to represent the cuff electrode (*Creating New Part Primitives*), please include copies of Java code that perform the FEM operations for the new parts. These sections of code should be from src/model/Part.java as "cases" (i.e., in switch-case statement) for new parts in the createCuffPartPrimitive() and createCuffPartInstance() methods.

Indicate how the cuff was placed on the nerve. Specifically, state the longitudinal placement of the cuff and how the cuff rotation was determined (e.g., the cuff rotation modes, or used a.tif to rotate the cuff to replicate in vivo cuff rotation).

4.4 Materials

Cite the original source for each conductivity value used to define materials as indicated in materials.json and/or model.json.

4.5 Domains

Report the model length and radius, along with a statement to justify your model's dimensions (i.e., convergence studies). Report the presence and dimensions of any cuff fill domains (e.g., saline, encapsulation, or mineral oil between the nerve and the cuff electrode).

4.6 Perineurium

Report if a thin-layer approximation or a finite thickness material was used to define the perineurium. If a thin-layer approximation was used, indicate how the thickness of the perineurium was determined (e.g., measured from histology, or previously published relationship between inner diameter and perineurium thickness).

4.7 Mesh

Indicate the method used to mesh the FEM in addition to the number of domain elements. Include a statement to justify your model's meshing parameters (i.e., convergence studies) (*Convergence Analysis Example*).

4.8 Solution

Indicate that the FEM was solved using Laplace's equation once for each contact delivering 1 mA of current. Indicate if the outer surfaces of the model were grounded or set to insulation.

4.9 HOC/MOD files

If using novel fiber model ultrastructure or channel mechanisms, please share the code required to implement it in ASCENT.

4.10 Waveform

State the stimulation waveform shape, timestep, and simulation time used. Be certain that the timestep chosen is short enough to simulate accurately fiber response to stimulation.

4.11 NEURON simulations

If computing thresholds of activation and/or block, state the search algorithm used (e.g., binary search) and the exit criteria (e.g., tolerance).

4.12 Analysis

State the methods and software tools used to analyze and present data.

CHAPTER

FIVE

CREATING MOCK MORPHOLOGY

MockSample is a Python class that manages the *data* and contains all operations to create binary masks of mock nerve morphology (i.e., nerve: `n.tif`, inners: `i.tif`, and scale bar: `s.tif`) to use as inputs to the pipeline.

The user defines the parameter values in `config/user/mock_samples/<mock_sample_index>.json` (with a template provided in `config/templates/mock_sample.json`). The mock sample morphology is then created using the JSON file by executing `"python run mock_morphology_generator <mock_sample_index>"` at the project root. The mock morphology generator uses the MockSample Python class to create binary images of the nerve, inner perineurium traces (fascicles), and the scale bar in `input/<NAME>/` (NAME is analogous to the "sample" parameter in ***Sample***, following the standard naming convention (*JSON Configuration Files*)), which allow the pipeline to function as if binary images of segmented histology were provided. The `<mock_sample_index>.json` file and the resulting segmented nerve morphology files are automatically saved in `input/<NAME>/`.

MockSample is Exceptionable, Configurable, and has instance attributes of "nerve" and a list "fascicles". After the MockSample class is initialized, a `mock_sample.json` file is added to the class instance. The `mock_morphology_generator.py` script configures an instance of the MockSample class using the first input argument, which references the index of a JSON file stored in `config/user/mock_samples/`.

In `mock_morphology_generator.py`, MockSample's methods `make_nerve()` and `make_fascicles()` are called to create ellipses for the nerve and fascicles in memory based on the parameters in the `mock_sample.json` file. MockSample's methods ensure that the fascicles have a minimum distance between each fascicle boundary and the nerve and between fascicle boundaries. For details on the parameters that define sample morphology using our mock nerve morphology generator, see *JSON Overview* for a description of mock_sample.json and *Mock Sample Parameters* for details of the syntax/data type of the key-value parameter pairs required to define a mock sample. Lastly, MockSample's `make_masks()` method is called on the class instance to create binary masks and save them as TIFs in the `input/<NAME>/` directory.

CHAPTER

SIX

PART PRIMITIVES AND CUSTOM CUFFS

6.1 Library of part primitives for electrode contacts and cuffs

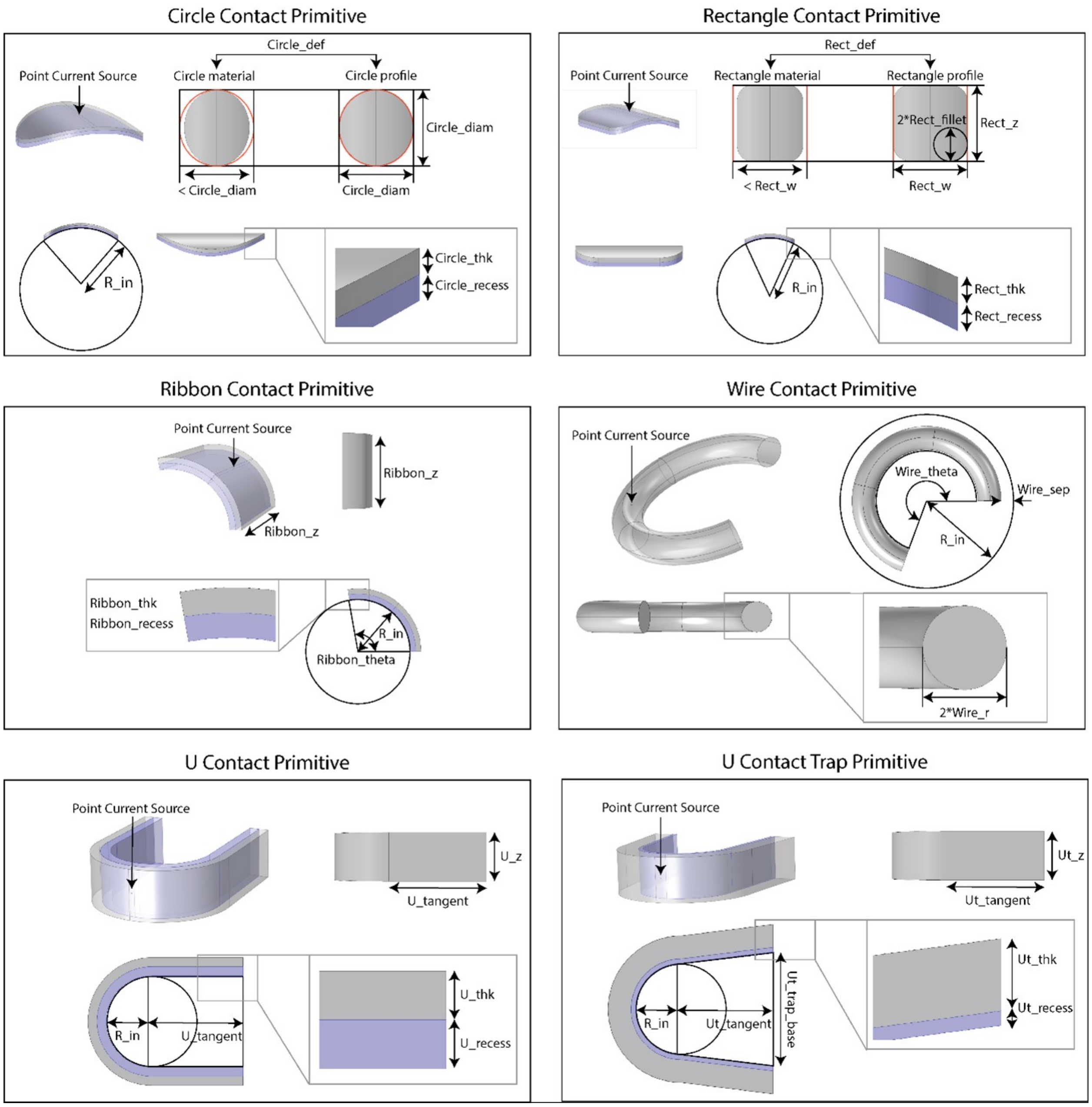


Figure A. Library of parts for electrode contacts. In the top two panels, "circle material" and "rectangle material" indicate that the contact's dimensions refer to the length of the materials, rather than the dimension of the contact when it is bent to wrap around the circular inner diameter of the cuff. The dimensions "_thk" refer to the contact thickness. The dimensions "_recess" refer to the thickness of the domain created if the contact is recessed into the cuff.

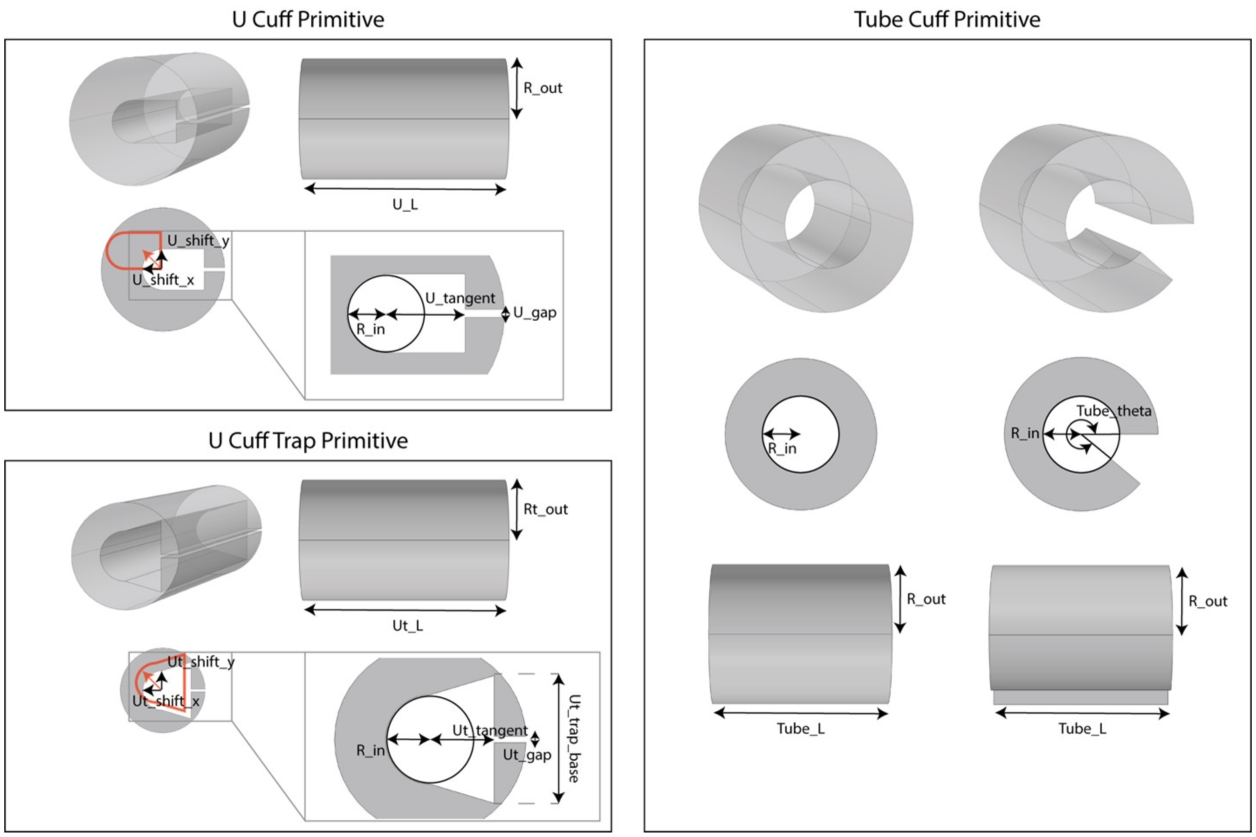


Figure B. Library of parts for cuffs. Parameters "U(t)_shift_x" and "U(t)_shift_y" change how the insulating material is centered around the inner diameter of the cuff as shown by the orange annotation.

6.2 Creating custom preset cuffs from instances of part primitives

The operations by which cuffs are added to the COMSOL "model" object are contained in the Java Part class (`src/model/Part.java`). A complete cuff design is defined by a JSON file (e.g., `Purdue.json`) stored in `config/system/cuffs/`, which we call a "preset" cuff, containing parameterizations and material assignments for part primitives that together represent an entire cuff electrode, including contact "recess" and "fill" (i.e., saline, mineral oil, or encapsulation tissue). The contents of the "preset" JSON file direct the Part class on which part primitives to add as well as their size, shape, placement, and material function (i.e., cuff "insulator", contact "conductor", contact "recess", and cuff "fill"). Fig 3A shows some examples of "preset" cuffs constructed from our library of COMSOL part primitives which are included in the pipeline repository in `config/system/cuffs/`. Users should not modify existing "preset" cuff files. Rather, a user should use our "preset" cuffs as a guide in constructing their own custom cuffs, which require a unique file name. Once a cuff is defined as a "preset" in `config/system/cuffs/` as its own JSON file, the user may choose to place the cuff on a nerve in COMSOL using the "preset" parameter in ***Model*** (*Model Parameters*).

We provide a COMSOL file in `examples/parts/sandbox.mph` that contains our library of "Geometry Parts" (i.e., part primitives) for users to assemble into their own cuffs in the COMSOL GUI. See *Creating New Part Primitives* for instructions on how to add new part primitives. After a part primitive is defined in the "Geometry Parts" node in the COMSOL GUI, under "Component 1", the user may secondary-click on the "Geometry 1" node -> "Parts" and select a part primitive. Importantly, the order of instantiated parts in the "Geometry 1" node matters for proper material assignment; the user must consider that when a new part occupies volume within a previously instantiated part, COMSOL will override the previous material assignment for the shared volume with the latter created part's material assignment. For example, if a user is modeling a cuff containing an embedded contact electrode (i.e., the contact surface is flush with the insulator's inner surface) and the entire cuff is bathed in saline (i.e., surgical pocket), the user would (1) add the part primitive for the saline cuff "fill" since it is the outer-most domain, (2) add the cuff "insulator" which would override its volume within the saline, and (3) add the contact "conductor" which would override the cuff insulator domain within the contact conductor. The order of instantiation of part primitives in COMSOL mirrors the order of the parts listed in the "preset" JSON file.

The part's required instantiation "Input Parameters" (in the "Settings" tab) have default values (found in the "Expression" dialogue boxes) that should be overridden and populated using parameter values to define the geometry of the part in your preset's implementation. The "Expression" dialogue boxes must contain parameter values already defined in a "Parameter Group" under the "Global Definitions" node (i.e., parameter names for either constants with units (e.g., "5 [um]") or mathematical relationships between other parameters (e.g., "parameter1 + parameter2")). The parameter values in the "Parameter Group" under the "Global Definitions" node are populated with the list of "params" in the "preset" cuff JSON file (explained in more detail below).

Once the user has succeeded in assembling their cuff in the COMSOL GUI from parameterized instantiations of parts in `examples/parts/sandbox.mph`, they are ready to create a new "preset" cuff JSON file in `config/system/cuffs/`. See the existing files in that directory for examples. The required elements of a "preset" cuff JSON file are shown in the skeleton structure below:

```
{
  "code": String
  "instances": [
    {
      "type": String,
      "label": Double,
      "def": {
         "parameter1": String // key is name of expected parameter in COMSOL
         ... // for all parameters (specific for part primitive "type")
      },
      "materials": [
         {
            "info": String,
            "label_index": Integer
         }
         ... // for all materials in a part
      ]
    }
    ... // for all instances
   ]
   "params": [
      {
         "name": String, // e.g., "parameter1_<code_value>" ... such as "pitch_Pitt"
         "expression": String,
         "description": String
      }
      ... // for all parameters
    ]
  },
  "expandable": Boolean,
  "fixed_point": String,
  "angle_to_contacts_deg": Double,
```

(continues on next page)

(continued from previous page)

```
  "offset": {
     "parameter1": Double,
        // parameter must be defined in "params" list, value is weight of
parameter value
        // (e.g., if radius of wire contact, need 2 to get contributing radial
distance
        // between nerve and cuff)
        ... // for all parameters informing how much "extra" space needed in cuff
   }
}
```

"code": The value (`String`) is a unique identifier for the parameters that are needed to define this cuff. All parameters in the `"params" [Object, ...]` will need to end with the characters of this code preceded by `"_"` (e.g., "code" = "Pitt", the "pitch" parameter for the separation between contacts would be `"pitch_Pitt"`).

"instances": The value is a list of JSON Objects, one for each part instance needed to represent the cuff. Within each part instance JSON Object, the user must define:

- `"type"`: The value (String) defines which known primitive to instantiate, which matches the switch-case in BOTH `Part.createCuffPartPrimitive()` AND `Part.createCuffPartInstance()` in Java (`src/model/Part.java`) behind the scenes.
- `"label"`: The value (String) defines the label that will show up in the COMSOL file to annotate the instance of the part primitive in the construction of your COMSOL FEM.
- `"def"`: The value (Object) contains all parameters required to instantiate the chosen part primitive. The key-value pairs will match the values entered in the COMSOL GUI for a part (i.e., "Settings" -> "Input Parameters" panel) in `examples/parts/sandbox.mph`.
  – Key-value pairs in this JSON Object will vary depending on the part primitive as defined in "type". For each parameter key, the value is a String containing a mathematical expression (of parameters) for COMSOL to evaluate.
- `"materials"`: List of JSON Objects for each material assignment in a part instance (*usually* this is just one material; contacts with recessed domains will have one material for the conductor and one material for the recessed domain as the part instance will create two separate domains with independent selections)
  – `"info"`: The value (String) is the function of the domain in the FEM (i.e., "medium", cuff "fill", cuff "insulator", contact "conductor", and contact "recess") that is used to assign material properties to a selected domain. The value will match a key in the "conductivities" JSON Object in ***Model***.
  – `"label_index"`: The value (Integer) corresponds to the index of the selection for the domain (in `im.labels`, defined independently for each primitive case in `Part.createCuffPartPrimitive()` – see code in `src/model/Part.java`) to be assigned to the material function. Note that `im.labels` are indexed starting at 0.

`"params"`: The value is a list of JSON Objects, one for each parameter used to define parameterizations of part primitives in COMSOL's Global Definitions. The structure of each JSON Object is consistent with the format of the dialogue boxes in each "Parameters" group (i.e., "Name", "Expression", "Description") where the parameters are populated in the COMSOL GUI:

- `"name"`: The value (String) is the name of the parameter.
- `"expression"`: The value (String) is the expression/constant with units that COMSOL will evaluate. Therefore, if the value is a constant, units wrapped in "[]" are required (e.g., "5 [um]"). If the value is an expression relating other parameters (that already are dimensioned with units) with known mathematical expressions (e.g., multiply `"*"`, divide `"/"`, add `"+"`, subtract `"-"`, exponent `"^"`, trigonometric formulas: `"sin()"`, `"cos()"`, `"tan()"`, `"asin()"`, `"acos()"`, `"atan()"`), do not add units after the expression.

- "description": The value (String) can be empty (i.e., “”) or may contain a description of the parameter such as a reference to the source (e.g., published patent/schematics) or a note to your future self.

"expandable": The value (Boolean) tells the system whether to expect the implementation of the cuff in COMSOL to be able to expand beyond the manufactured resting cuff diameter to fit around a nerve. For a cuff to be expandable, it must be constructed from part primitives that have been parameterized to expand as a function of "R_in". See config/system/cuffs/Purdue.json for an example of an expandable cuff. Expandable cuffs should be parameterized such that the contact length remains constant.

"fixed_point": The value (String) defines which point on the cuff remains fixed when expanding. Note that these are not behaviors, this option will change nothing regarding how the cuff is generated. This parameter is descriptive only, indicating how the cuff is parameterized based on the part primitives. This parameter is to inform the cuff shift algorithm on how it should account for cuff expansion. Currently two options are implemented:

1. "center": In this case, the center of the contact always remains at the same angle when expanding.
2. "clockwise_end": In this case, the clockwise end of either the cuff, or the contact (both will work correctly) remains fixed. Note, this option assumes that the fixed point for the clockwise end is at theta = 0.

"angle_to_contacts_deg": The value (Double, units: degrees) defines the angle to the contact point of the nerve on the inside of the cuff of the cuff (measured counterclockwise from the +x-axis before any rotation/deformation of the cuff).

"offset": The JSON Object contains keys that are names of parameters defined in the list of “params” in this “preset” cuff file. If the inner diameter of the cuff must expand to allow for additional distance between the nerve and "R_in" (the inner surface of the cuff insulator), the user adds key-value pairs for the offset buffer here. For each known parameter key, the user sets the value (Double) to be the multiplicative factor for the parameter key. The list of key-value pairs can be empty, as is the case for most cuffs. Once offset is added to a cuff, the values are automatically used in Runner’s compute_cuff_shift() method.

For example usage of this functionality, see config/system/cuffs/Purdue.json as replicated below:

```
"offset": {
   "sep_wire_P": 1, // sep_wire_P is the separation between the wire contact and the
                    // inner diameter of the cuff
   "r_wire_P": 2    // r_wire_P is the radius of the circular cross section of the
                    // wire contact (i.e., half of the wire’s gauge)
}

// the above JSON Object adds an offset buffer between the nerve and the inner
// diameter of the cuff of : (1*sep_wire_P) + (2*r_wire_P)
```

6.3 Creating new part primitives

Though we provide a library of part primitives to assemble representations of many cuff electrodes as shown in examples/parts/sandbox.mph, users may find it necessary to add their own part primitives for representing their custom cuff electrode designs. Use the following instructions as a guide and link to resources for creating a new part primitive.

The COMSOL GUI has a “Geometry Parts” node under the “Global Definitions”. The pipeline adds part primitives—i.e., the geometry of different pieces of cuff electrodes (e.g., contacts (*ASCENT Part Primitives* Figure A), insulators (*ASCENT Part Primitives* Figure B), cuff fill (e.g., encapsulation tissue, mineral oil, saline), or medium (e.g., surrounding muscle, fat)—as “parts” under “Geometry Parts”. Their resulting volumes (domains), surfaces, and points (used for point current sources) are added to the list of “cumulative selections” which are later used to assign appropriate mesh settings, material properties, and boundary conditions.

1. Create and label your part. Open up `examples/parts/sandbox.mph`, secondary-click on "Geometry Parts", choose "3D Part". Give your part an appropriate label as it will later be used in creating "preset" cuff JSON files and as a flag for the primitive in our Java Part class (`src/model/Part.java`).

2. Define your part's geometry. Secondary-click on the new part under "Geometry Parts" to add the geometry operations required to construct your cuff (e.g., "Block", "Cone", "Cylinder", "Sphere", "More Primitives", "Work Plane"). See the operations under the other part primitives in `sandbox.mph` as a guide in creating your geometry, as well as COMSOL's documentation (in particular their "Appendix A – Building a Geometry") and "Introductory Video Series on How to Build Geometries in COMSOL". In creating your geometry, carefully label each operation to improve readability of your geometry operations. Taking care to label the operations will not only help to communicate to other users and your future self what operations must occur, but also will help you in cleaning up your Java code in the Part class (part of step 6). The dimensions and locations of part geometries should refer to "Input Parameters" or parameters stored in a parameters group under "Global Definitions".

3. Assign each feature of interest (i.e., volumes/domains, surfaces, points) to a "Cumulative selection". The final form of domains that you will want to assign to a material property need to be assigned to a "Cumulative selection", which can be found for the final geometry operation under the "Settings" tab. In "Settings", under the "Contribute to" drop down menu within the "Selections of Resulting Entities" section, the first time you refer to a cumulative selection you will need to click the "New" button and type in the name of the selection. Again, take care to give an informative label. Our convention, though not required, is to make the cumulative selection names in all capitals to improve code readability.

4. Compact `sandbox.mph` file history. File -> Compact History removes any operations you may have tested but did not ultimately use to create your part.

5. Export Java code. Go to File -> Save As, give a meaningful path/file name, and change the file type to "Model File for Java (`*.java`)".

6. Add the operations for your part primitive to the `Part.createCuffPartPrimitive()` method in Java (`src/model/Part.java`). With the text editor of your choice, open the newly created `*.java` file. Toward the bottom of the file, the operations you just performed in COMSOL are contained in a block of code. Performing a "`Command+F`" (on Mac) or "`Control+F`" (on Windows) for your new part primitive's name that you gave in the GUI (from step 1) should take you to the first line of the code block of interest, which looks like:

```
"model.geom("part<#>").label(<your part's name>);
```

All subsequent lines starting with "`model.geom("part<#>")`" are of interest. Copy them to your clipboard. With `Part.createEnvironmentPartPrimitive()` as a guide (since it is by far the simplest and most contained "primitive" – in reality it is just a cylinder contributed to the "MEDIUM" cumulative selection), add your lines to `Part.createCuffPartPrimitive()`.

- Add a "case" in the switch-case (e.g., case "`TubeCuff_Primitive`"). Within this case-block, all operations for the new primitive will be added.
- For each line in your code (copied from the exported `*.java` file) that begin with "`model.geom.("part<#>").inputParam().set("<my_parameter>", "<default_value>")`", at the top of your new case-block, add the following line to set the "Input Parameters" you established in the COMSOL GUI:
  - `mp.set("<my_parameter>", "<default_value>")`
- Still looking at `Part.createEnvironmentPartPrimitive()` as an example, now create your list of "selections" in "im.labels" which are the lines that follow after the "Input Parameters" are defined. Then add the for loop that loops over the `im.labels [String, ...]` adding them to COMSOL's selections. These lines will look like:

```
im.labels = new String[] {
"<MY_CUMULATIVE SELECTION_1>", // note: This is index 0
"<MY_CUMULATIVE SELECTION_2>", // note: This is index 1
```

(continues on next page)

(continued from previous page)

```
...
}
```

```
for (String cselLabel : im.labels) {
model.geom(id).selection().create(im.next("csel", cselLabel),
"CumulativeSelection").label(cselLabel)
}
```

- The lines that follow the Cumulative Selection labeling add the geometry features of the COMSOL part which COMSOL has also conveniently exported for you in the `*.java` file. See *Java Utility Classes* for an explanation of our Java `IdentifierManager` utility class that we created to abstract away from COMSOL's indexing system to improve code readability. Our `"IdentifierManager"` class enables the user to access previously defined selection tags by an informative label programmatically. `Part.createEnvironmentPartPrimitive()` and `Part.createCuffPartPrimitive()` are also great working examples of how we use the COMSOL plugin in an IDE to clean up the code (e.g., creating a COMSOL "GeomFeature" to shorten the length of each line).
- End the case for your new part primitive with a `"break;"` (this is very important!)

7. Add the operations for your part primitive to the `Part.createCuffPartInstance()` method (`src/model/Part.java`).

- Create a `List[String]` containing the "Input Parameters" you established in the COMSOL GUI. These lines will look like:

```
String[] myPrimitiveParameters = {
"<my_parameter1>",
"<my_parameter2>",
... // for all Input Parameters
}
```

- Add a for loop that adds all the "Input Parameters" to the part instance. These lines will look like:

```
for (String param : myPrimitiveParameters) {
partInstance.setEntry("inputexpr", param, (String) itemObject.get(param));
}
```

- Our primitives have an additional (optional) section for selection imports. Each defined selection used in your geometry operations will be visible in the "Contribute to" drop-down menu unless you toggle each selection "off". An example of how to "clean up" your selections imported is shown below:

// imports

// so that the program only imports selections that are used `partInstance.set("selkeepnoncontr", false);`

// to selectively import the DOMAIN for whatever selection index 0 is in myLabels (defined in `im.labels` in // `createCuffPartPrimitive()`). To exclude, "off" instead of "on". `partInstance.setEntry("selkeepdom", instanceID + "_" + myIM.get(myLabels[0] + ".dom", "on"));`

// to selectively import the BOUNDARY for whatever selection index 0 is in myLabels (defined in `im.labels` // in `createCuffPartPrimitive()`). To exclude, "off" instead of "on". `partInstance.setEntry("selkeepbnd", instanceID + "_" + myIM.get(myLabels[0] + ".bnd", "on"));`

// to selectively import the POINT for whatever selection index 0 is in myLabels (defined in `im.labels` // in `createCuffPartPrimitive()`). To exclude, "off" instead of "on". `partInstance.setEntry("selkeeppnt", instanceID + "_" + myIM.get(myLabels[0] + ".pnt", "on"));`

- End the case for your new part instance with a "break;" (this is very important!)

8. (Step is optional but recommended). Add your new part to `examples/parts/sandbox.mph`. Simply save your `sandbox.mph` file as a `*.mph`, if you have not already, for future ability to assemble cuffs using your new part in the COMSOL GUI.

CHAPTER

SEVEN

CONVERGENCE ANALYSIS EXAMPLE

*Note: The following convergence analysis was performed for a rat cervical vagus nerve and is provided as an illustrative example.

The outer boundaries of the model (assigned to electrical ground) were initially set to dimensions that are known to be excessively large in both length (100 mm) and radius (20 mm) (Figure A). For testing of convergence of the model length and diameter, we carefully controlled the mesh parameters such that any changes in the threshold to excite model nerve fibers positioned in the finite element model are attributed to changes in model dimensions and not to a change in mesh density. First, we used an excessive number of domain mesh elements. Second, to maintain consistent meshing in the area of interest despite a change in geometry of the outer boundary of the model, the proximal region of the model surrounding the nerve and electrode (i.e., "proximal", which is the full length of the nerve) was assigned its own mesh entirely independent from the outer boundary region (i.e., "distal").

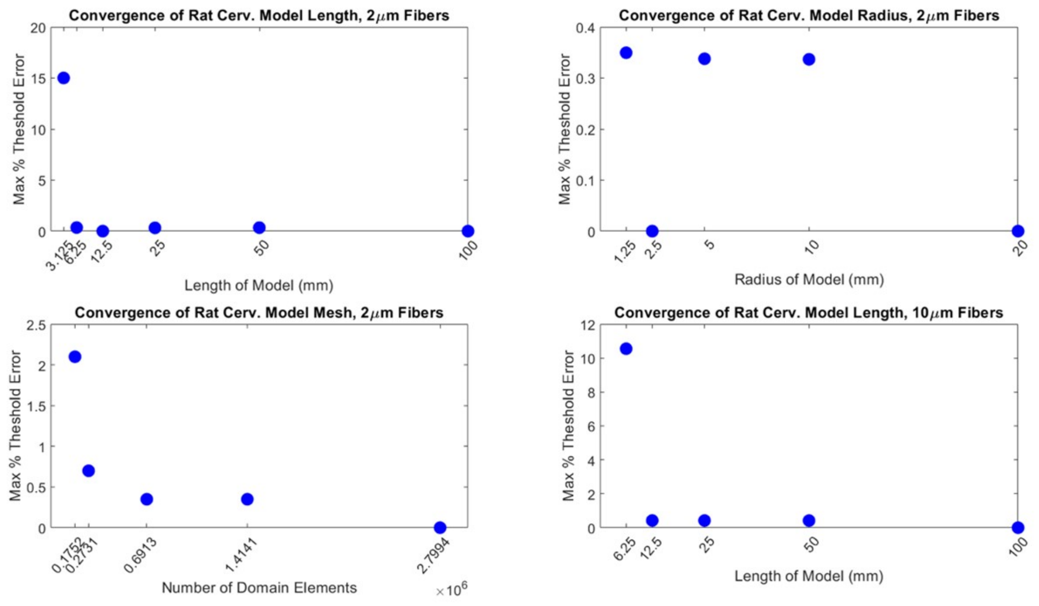


Figure A. Convergence of FEM model dimensions and mesh density for 2 and 10 m myelinated MRG fibers modeled in a rat cervical vagus nerve.

For different model sizes and mesh densities, we determined excitation thresholds for 100 randomly placed 2 µm diameter myelinated model axons in the nerve. First, with the radius of the model held constant at 20 mm, we halved the length of the model until there was a change in threshold of greater than 2% compared to the 100 mm-long model. The resulting length of the model was chosen to be 6.25 mm, as the next shortest length (3.125 mm) had at least one fiber with a threshold change of 15%, which was above our tolerance of 2%. Second, with the length of the model held constant at 6.25 mm, the radius of the model was halved until there was a change in threshold of greater than 2% compared to the 20 mm-radius model. The resulting radius of the model was chosen to be 1.25 mm. The radius of the model could not be reduced further without interfering with the electrode geometry. We then reduced the number of mesh elements until there was a 2% change in thresholds for activation of model nerve fibers. Meshing parameters that resulted in ~273,000 domain elements for the 1.25 mm radius and 6.25 mm length model were chosen for further studies.

Lastly, the length was again converged with 10 µm diameter myelinated model axons using the previously found model radius and mesh density to ensure that the model length incorporated enough nodes of Ranvier for convergence of threshold for all native fiber diameters. A conservative estimate for the largest myelinated fiber diameter found in the cervical level of a rat vagus nerve is 10 µm [de Alcântara *et al.*, 2008]. The largest diameter nerve fiber native to this region is of interest in our convergence studies because it will have the fewest number of nodes per unit length (based on INL=100*D). We found that the FEM length needed to be longer for 10 µm diameter myelinated fibers (12.5 mm) than 2 µm diameter myelinated fibers (6.25 mm).

CHAPTER

EIGHT

TROUBLESHOOTING GUIDE

8.1 General Troubleshooting Steps

- Check your geometry (ascent/samples/sample_index/models/model_index/debug_geom.mph)
- Check your sample (ascent/samples/sample_index/slides/cassette_index/slide_index/masks)

8.2 Installation Issues

- NEURON
  - Issue: Sometimes NEURON installation will complete but NEURON will not be installed
  - Solution: Run the installer in Windows 8 compatibility mode

8.3 Pipeline Issues

- Sample
  - Issue: Small white pixel islands cause the pipeline to fail, or the pipeline does not fail but very small elements are present as fascicles that reflect segmentation errors
  - Solution: increase value of image_preprocessing>object_removal_area in sample.json
- Model
  - Issue: fascicle borders appear ragged/wavy in model geometry
  - Solutions:
    - Increase inner_interp_tol and/or outer_interp_tol in sample.json (depending on whether the Issue exists for outers or inners)
    - If your mask is very pixelated, increase smoothing>fascicle distance in sample.json
  - Issue: The pipeline solves one basis, then errors.
  - Solution: In COMSOL, disable automatic saving of recovery files.

8.4 COMSOL Issues

- Issue: fiber xy-location(s) fall outside of the solution, which results in error while extracting potentials.- Error log:- `Exception: com.comsol.util.exceptions.FlException: Internal numerical error`- `Messages: Internal error in numerical routines.`-Output log:- `Intel MKL Error: Parameter 7 was incorrect on entry to DGELS.`
- Solution: Increase parameter in Sim -> "fibers" -> "xy_trace_buffer". Note: this is assuming that "endo_only_solution" in Run config is true.

8.5 NEURON Issues

- Issue: Compiling NEURON files with `python submit.py` results in an error, `mpicc: command not found`
- Solution: Install open-mpi 2.0 and add to path

8.6 Python Issues

- Issue: python output prints out of order with java output
- Solution: pass the -u flag (i.e. python -u run pipeline), which turns off output buffering

CHAPTER

NINE

CODE HIERARCHY

9.1 Python Classes

9.1.1 Python classes for representing nerve morphology (Sample)

The nerve cross section includes the outer nerve trace (if present; not required for monofascicular nerves) and, for each fascicle, either a single "inner" perineurium trace or both "inner" and "outer" perineurium traces. We provide automated control to correct for tissue shrinkage during histological processes [Boyd and Kalu, 1979] (*Sample Parameters*). Morphology metrics (e.g., nerve and fascicle(s) cross-sectional areas and centroids, major and minor axis lengths, and rotations of the best-fit ellipses) are automatically reported in ***Sample*** (*Sample Parameters*).

Trace

Trace is the core Python class for handling a list of points that define a closed loop for a tissue boundary in a nerve cross section (see "Tissue Boundaries" in Fig 2). Trace has built-in functionality for transforming, reporting, displaying, saving, and performing calculations on its data contents and properties. Python classes Nerve, Fascicle, and Slide are all special instances or hierarchical collections of Trace.

A Trace requires inputs of a set of (x,y)-points that define a closed loop and an exceptions JSON configuration file. The z-points are assumed to be '0'. The Trace class already provides many built-in functionalities, but any further user-desired methods needed either to mutate or access nerve morphology should be added to the Trace class.

Trace uses the *OpenCV* [Bradski, 2000], *Pyclipper* [Chalton and others, 2015–], and *Shapely* [Gillies and others, 2007–] Python packages to support modifier methods (e.g., for performing transformations):

- `scale()`: Used to assign dimensional units to points and to correct for shrinkage of nerve tissues during processing of histology.
- `rotate()`: Performs a rigid rotational transformation of Trace about a point (positive angles are counter-clockwise and negative are clockwise).
- `shift()`: performs a 2D translational shift to Trace (in the (x,y)-plane, i.e., the sample cross section).
- `offset()`: Offsets Trace's boundary by a discrete distance from the existing Trace boundary (non-affine transformation in the (x,y)-plane, i.e., the sample cross section).
- `pymunk_poly()`: Uses *Pymunk* to create a body with mass and inertia for a given Trace boundary (used in `deform()`, the fascicle repositioning method, from the Deformable class).
- `pymunk_segments()`: Uses *Pymunk* to create a static body for representing intermediate nerve boundaries (used in `deform()`, the fascicle repositioning method, from the Deformable class).

Trace also contains accessor methods:

- `within()`: Returns a Boolean indicating if a Trace is completely within another Trace.

- `intersects()`: Returns a Boolean indicating if a Trace is intersecting another Trace.
- `centroid()`: Returns the centroid of the best fit ellipse of Trace.
- `area()`: Returns the cross-sectional area of Trace.
- `random_points()`: Returns a random list of coordinates within the Trace (used to define axon locations within the Trace).

Lastly, Trace has a few utility methods:

- `plot()`: Plots the Trace using formatting options (i.e., using the `plt.plot` format, see Matplotlib documentation for details).
- `deepcopy()`: Fully copy an instance of Trace (i.e., copy data, not just a reference/pointer to original instance).
- `write()`: Writes the Trace data to the provided file format (currently, only COMSOL's sectionwise format—ASCII with .txt extension containing column vectors for x- and y-coordinates—is supported).

Nerve

Nerve is the name of a special instance of Trace reserved for representing the outer nerve (epineurium) boundary. It functions as an alias for Trace. An instance of the Nerve class is created if the `"NerveMode"` in ***Sample*** ("nerve") is "PRESENT" (*Sample Parameters*)"

Fascicle

Fascicle is a class that bundles together instance(s) of Trace to represent a single fascicle in a slide. Fascicle can be defined with either (1) an instance of Trace representing an outer perineurium trace and one or more instances of Trace representing inner perineurium traces, or (2) an inner perineurium trace that is subsequently scaled to make a virtual outer using Trace's methods `deepcopy()` and `offset()` and the perineurium thickness defined by the `"PerineuriumThicknessMode"` in ***Sample*** (`"ci_perineurium_thickness"`) (*Sample Parameters*). Upon instantiation, Fascicle automatically validates that each inner instance of Trace is fully within its outer instance of Trace and that no inner instance of Trace intersects another inner instance of Trace.

Fascicle contains methods for converting a binary mask image of segmented fascicles into instances of the Fascicle class. The method used depends on the contents of the binary image inputs to the pipeline as indicated by the `"MaskInputMode"` in ***Sample*** (`"mask_input"`) (i.e., `INNER_AND_OUTER_SEPARATE`, `INNER_AND_OUTER_COMPILED`, or `INNERS`). For each of the mask-to-Fascicle conversion methods, the OpenCV Python package finds material boundaries and reports their nested hierarchy (i.e., which inner Traces are within which outer Traces, thereby associating each outer with one or more inners). The methods are expecting a *maximum* hierarchical level of 2: one level for inners and one level for outers.

- If separate binary images were provided containing contours for inners (`i.tif`) and outers (`o.tif`), then the `"MaskInputMode"` in ***Sample*** (`"mask_input"`, *Sample Parameters*) is `INNER_AND_OUTER_SEPARATE`.
- If a single binary image was provided containing combined contours of inners and outers (`c.tif`), then the `"MaskInputMode"` in ***Sample*** (`"mask_input"`, *Sample Parameters*) is `INNER_AND_OUTER_COMPILED`.
- If only a binary image was provided for contours of inners (`i.tif`), the `"MaskInputMode"` (`"mask_input"`, *Sample Parameters*) in ***Sample*** is `INNERS`.

In all cases, the Fascicle class uses its `to_list()` method to generate the appropriate Traces. Additionally, Fascicle has a `write()` method which saves a Fascicle's inner (one or many) and outer Traces to files that later serve as inputs for COMSOL to define material boundaries in a nerve cross section (sectionwise file format , i.e., ASCII with `.txt` extension containing column vectors for x- and y-coordinates). Lastly, Fascicle has a `morphology_data()` method which uses Trace's `area()` and `ellipse()` methods to return the area and the best-fit ellipse centroid, axes, and rotation of each outer and inner as a JSON Object to ***Sample*** (*Sample Parameters*).

Slide

The Slide class represents the morphology of a single transverse cross section of a nerve sample (i.e., nerve and fascicle boundaries). An important convention of the pipeline is that the nerve is always translated such that its centroid (i.e., from best-fit ellipse) is at the origin (x,y,z) = (0,0,0) and then extruded in the positive (z)-direction in COMSOL. Slide allows operations such as translation and plotting to be performed on all Nerve and Fascicle Traces that define a sample collectively.

To create an instance of the Slide class, the following items must be defined:

- A list of instance(s) of the Fascicle class.
- `"NerveMode"` from ***Sample*** ("nerve") (i.e., `PRESENT` as in the case of nerves with epineurium (`n.tif`) or `NOT_PRESENT` otherwise, (*Sample Parameters*)).
- An instance of the Nerve class if `"NerveMode"` is `PRESENT`.
- A Boolean for whether to reposition fascicles within the nerve from `"ReshapeNerveMode"` in ***Sample*** (*Sample Parameters*).
- A list of exceptions.

The Slide class validates, manipulates, and writes its contents.

- In Slide's `validation()` method, Slide returns a Boolean indicating if its Fascicles and Nerve Traces are overlapping or too close to one another (based on the minimum fascicle separation parameter in ***Sample***).
- In Slide's `smooth_traces()` method, Slide applies a smooth operation to each Trace in the slide, leveraging the offset method, and applying smoothing based on `smoothing` parameters. defined in ***Sample*** (*Sample Parameters*).
- In Slide's `move_center()` method, Slide repositions its contents about a central coordinate using Trace's `shift()` method available to both the Nerve and Fascicle classes (by convention, in ASCENT this is (x,y) = (0,0)).
- In Slide's `reshaped_nerve()` method, Slide returns the deformed boundary of Nerve based on the `"ReshapeNerveMode"` in ***Sample*** (`"reshape_nerve"`, (*Sample Parameters*)) (e.g., CIRCLE).
- Using the methods of Nerve and Fascicle, which are both manifestations of Trace, Slide has its own methods `plot()`, `scale()`, and `rotate()`.
- Slide has its own `write()` method which determines the file structure to which the Trace contours are saved to file in `samples/<sample index>/slides/`.

Note that the sample data hierarchy can contain more than a single Slide instance (the default being 0 as the cassette index and 0 as the section index, hence the 0/0 seen in ASCENT Data Hierarchy *Figure A) even though the pipeline data processing assumes that only a single Slide exists. This will allow the current data hierarchy to be backwards compatible if multi-Slide samples are processed in the future.*

Map

Map is a Python class used to keep track of the relationship of the longitudinal position of all Slide instances for a Sample class. At present, the pipeline only supports models of nerves with constant cross-sectional area, meaning only one Slide is used per FEM, but this class is implemented for future expansion of the pipeline to construct three-dimensional nerve models with varying cross section (e.g., using serial histological sections). If only one slide is provided, Map is generated automatically, and the user should have little need to interact with this class.

Sample

The Sample class is initialized within Runner's `run()` method by loading ***Sample*** and ***Run*** configurations (*JSON Configuration Files*). First, Sample's `build_file_structure()` method creates directories in `samples/` and populates them with the user's file inputs from `input/<NAME>/`; the images are copied over for subsequent processing, as well as for convenience in creating summary figures. Sample then uses its `populate()` method to construct instances of Nerve and Fascicle in memory from the input sample morphology binary images (see Fascicle class above for details). Sample's `populate()` method packages instances of Nerve and Fascicle(s) into an instance of Slide.

Sample's `get_factor()` method obtains the ratio of microns/pixel for the input masks, either utilizing a scale bar image, or an explicit scale factor input depending on the user's `ScaleInputMode` defined in ***Sample*** (*Sample Parameters*).

Sample's `scale()` method is used to convert Trace points from pixel coordinates to coordinates with units of distance based either on the length of the horizontal scale bar as defined in ***Sample*** (micrometers) and the width of the scale bar (pixels) in the input binary image (`s.tif`), or on the explicitly specified scale ratio defined in ***Sample*** (*Sample Parameters*). If using a scale bar for scale input, it must be a perfectly horizontal line. Sample's scale() method is also used within `populate()` to correct for shrinkage that may have occurred during the histological tissue processing. The percentage increase for shrinkage correction in the slide's 2D geometry is stored as a parameter "shrinkage" in ***Sample*** (*Sample Parameters*). Additionally, Slide has a `move_center()` method which is used to center Slide about a point within `populate()`. Note that Sample is centered with the centroid of the best-fit ellipse of the outermost Trace (Nerve if `"NerveMode"` in ***Sample*** ("nerve") is `"PRESENT"`, outer Trace if `"NerveMode"` is `"NOT_PRESENT"`, (*Sample Parameters*)) at the origin (0,0,0). Change in rotational or translational placement of the cuff around the nerve is accomplished by moving the cuff and keeping the nerve position fixed (*Cuff Placement on the Nerve*).

Sample's `im_preprocess()` method performs preprocessing operations on the binary input masks based on the parameters given under `preprocess` in ***Sample*** (*Sample Parameters*).

Sample's `io_from_compiled()` generates outers (o.tif) and inners (i.tif) from the (compiled) c.tif mask if `"MaskInputMode"` is `INNER_AND_OUTER_COMPILED`. These generated masks are then used in Fascicle's `to_list()` method.

Sample's `generate_perineurium()` method is used to generate the perineurium for fascicle inners in the case where `"MaskInputMode"` = `INNERS`. This leverages Trace's `offset()` method, and fits the generated perineurium based on `ci_perineurium_thickness` defined in ***Sample*** (*Sample Parameters*).

Sample's `populate()` method also manages operations for saving tissue boundaries of the Sample (Nerve and Fascicles) to CAD files (`slides/#/#/sectionwise2d/`) for input to COMSOL with Sample's `write()` method.

Sample's `output_morphology_data()` method collects sample morphology information (area, and the best-fit ellipse information: centroid, major axis, minor axis, and rotation) for each original Trace (i.e., Fascicle inners and outers, and Nerve) and saves the data under "Morphology" in ***Sample.***

Lastly, since Sample inherits `Saveable`, Sample has access to the `save()` method which saves the Python object to file.

Deformable

If `"DeformationMode"` in ***Sample*** ("deform") is set to `NONE`, then the Deformable class takes no action (*Sample Parameters*). However, if `"DeformationMode"` in ***Sample*** is set to `PHYSICS`, then Deformable's `deform()` method simulates the change in nerve cross section that occurs when a nerve is placed in a cuff electrode. Specifically, the outer boundary of a Slide's Nerve mask is transformed into a user-defined final geometry based on the `"ReshapeNerveMode"` in ***Sample*** (i.e., `CIRCLE`) while maintaining the cross-sectional area. Meanwhile, the fascicles (i.e., outers) are repositioned within the new nerve cross section in a physics-based way using Pymunk [Blomqvist, 2007], a 2D physics library, in which each fascicle is treated as rigid body with no elasticity as it is slowly "pushed" into place by both surrounding fascicles and the nerve boundary (Figure A).

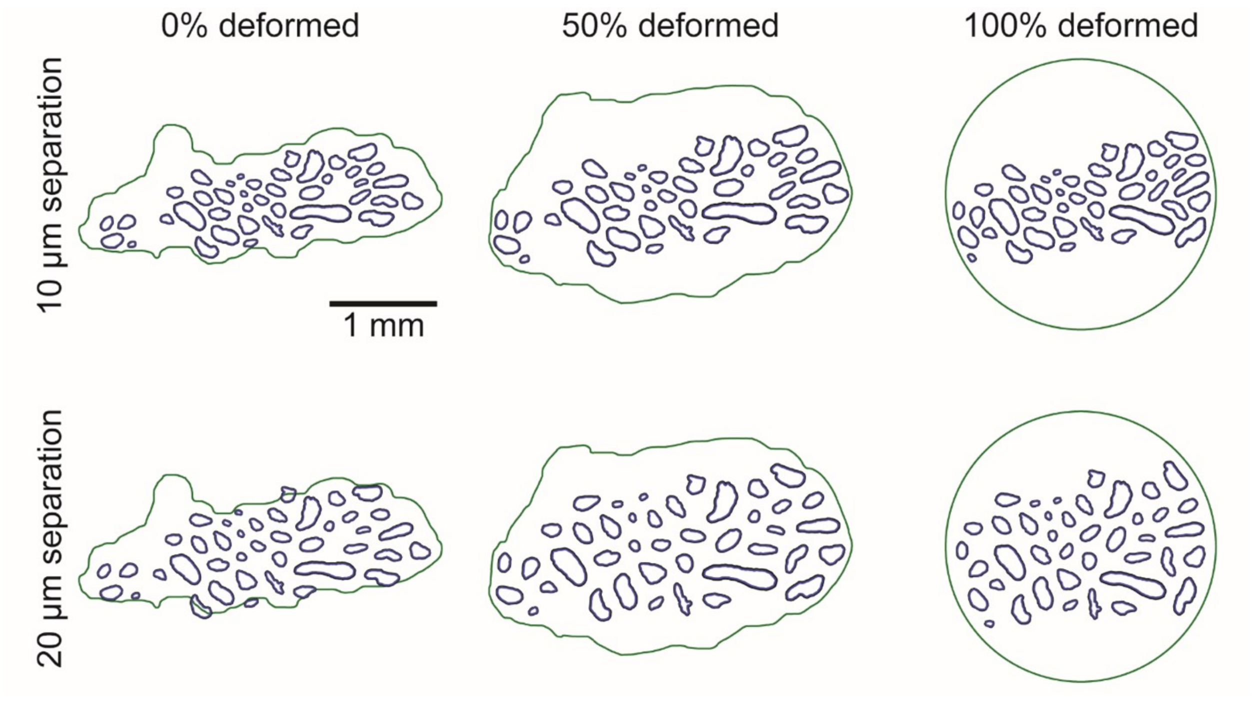


Figure A. Snapshots at 0%, 50%, and 100% (left-to-right) of the deformation process powered by the pygame package [Shinners, 2011–]. The deformation process is shown for two minimum fascicle separation constraints: 10 µm (top row) and 20 µm (bottom row). The geometry at 0% deformation is shown after the fascicles have been spread out to the minimum separation constraint.

The `deform()` method updates the nerve boundary to intermediately-deformed nerve traces between the nerve's `"boundary_start"` (i.e., the Trace's profile in segmented image) and `"boundary_end"` (i.e., the Trace's profile after accommodation to the cuff's inner diameter, which is determined by the "ReshapeNerveMode" (`"reshape_nerve"`, *Sample Parameters*) while the fascicle contents are allowed to rearrange in a physics-space. By default, all fascicles have the same "mass", but their moment of inertia is calculated for each fascicle based on its geometry (see Trace's `pymunk_poly()` method). Each fascicle is also assigned a "friction coefficient" of 0.5 as well as a "density" of 0.01. These measurements are mostly important as they relate to one another, not as absolute values. Importantly, we set the elasticity of all the fascicles to 0, so all kinetic energy is absorbed, and fascicles only move if they are directly pushed by another fascicle or by the nerve barrier. In Sample's `populate()` method, the final fascicle locations and rotations returned by the `deform()` method are then applied to each fascicle using the Fascicle class's `shift()` and `rotate()` methods.

Deformable's convenience constructor, `from_slide()`, is automatically called in Sample's `populate()` method, where a Slide is deformed to user specification. The `from_slide()` method takes three input arguments: The Slide object from which to construct the current Deformable object, the `"ReshapeNerveMode"` (e.g., CIRCLE, *Sample Parameters*), and the minimum distance between fascicles. If only inners are provided, virtual outers interact during nerve deformation to account for the thickness of the perineurium. Each inner's perineurium thickness is defined by the `"PerineuriumThicknessMode"` in ***Sample*** (`"ci_perineurium_thickness"`, *Sample Parameters*), which specifies the linear relationship between inner diameter and perineurium thickness defined in `config/system/ci_peri_thickness.json` (*Contact Impedance*). Deformable's `from_slide()` method uses Deformable's `deform_steps()` method to calculate the intermediately-deformed nerve traces between the `boundary_start` and the `boundary_end`, which contain the same number of points and maintain nerve cross-sectional area. The `deform_steps()` method maps points between `boundary_start` and `boundary_end` in the following manner. Starting from the two points where the major axis of the Nerve's best-fit ellipse intersects `boundary_start` and `boundary_end`, the algorithm matches consecutive `boundary_start` and `boundary_end` points and calculates the vectors between all point pairs. The `deform_steps()` method then returns a list of intermediately-deformed nerve traces between the `boundary_start` and `boundary_end` by adding linearly-spaced portions of each point pair's vector to `boundary_start`. Also note that by defining `"deform_ratio"` (value between 0 and 1) in ***Sample***, the user can optionally indicate a partial deformation of the Nerve (*Sample Parameters*). In the case where `"deform_ratio"` is set to 0, minimum fascicle separation will still be enforced, but no changes to the nerve boundary will occur.

Enforcing a minimum fascicle separation that is extremely large (e.g., 20 μm) can cause inaccurate deformation, as fascicles may be unable to satisfy both minimum separation constraints and nerve boundary constraints.

To maintain a minimum distance between adjacent fascicles, the Deformable's `deform()` method uses Trace's `offset()` method to perform a non-affine scaling out of the fascicle boundaries by a fixed distance before defining the fascicles as rigid bodies in the `pygame` physics space. At regular intervals in physics simulation time, the nerve boundary is updated to the next Trace in the list of intermediately-deformed nerve traces created by `deform_steps()`. This number of Trace steps defaults to 36 but can be optionally set in ***Sample*** with the `"morph_count"` parameter by the user (*Sample Parameters*). It is important to note that too few Trace steps can result in fascicles lying outside of the nerve during deformation, while too many Trace steps can be unnecessarily time intensive. We've set the default to 36 because it tends to minimize both aforementioned issues for all sample sizes and types that we have tested.

The user may also visualize nerve deformation by setting the `"deform_animate"` argument to true in `sample.populate()` (called in Runner's `run()` method) (*Sample Parameters*). Visualizing sample deformation can be helpful for debugging but increases computational load and slows down the deformation process significantly. When performing deformation on many slides, we advise setting this flag to false.

9.1.2 Simulation

(Pre-Java)

The user is unlikely to interface directly with Simulation's `resolve_factors()` method as it operates behind the scenes. The method searches through ***Sim*** for lists of parameters within the "fibers" and "waveform" JSON Objects until the indicated number of dimensions ("n_dimensions" parameter in ***Sim***, which is a handshake to prevent erroneous generation of NEURON simulations) has been reached. The parameters over which the user has indicated to sweep in ***Sim*** are saved to the Simulation class as a dictionary named "factors" with the path to each parameter in ***Sim***.

The required parameters to define each type of waveform are in *Sim Parameters*. The Python Waveform class is configured with ***Sim***, which contains all parameters that define the Waveform. Since FEMs may have frequency-dependent conductivities, the parameter for frequency of stimulation is optionally defined in ***Model*** (for frequency-dependent material conductivities), but the pulse repetition frequency is defined in ***Sim*** as `"pulse_repetition_freq"`. The `write_waveforms()` method instantiates a Python Waveform class for each `"wave_set"` (i.e., one combination of stimulation parameters).

(Post-Java)

The unique combinations of ***Sim*** parameters are found with a Cartesian product from the listed values for individual parameters in ***Sim***: Waveforms Src_weights Fibersets. The pipeline manages the indexing of simulations. For ease of debugging and inspection, into each `n_sim/` directory we copy in a modified "reduced" version of ***Sim*** with any lists of parameters replaced by the single list element value investigated in the particular `n_sim/` directory.

The Simulation class loops over ***Model*** and ***Sim*** as listed in ***Run*** and loads the Python `"sim.obj"` object saved in each simulation directory `(sims/\<sim index\>/)` prior to Python's `handoff()` to Java. Using the Python object for the simulation loaded into memory, the Simulation class's method `build_n_sims()` loops over the `master_product_index` (i.e., waveforms (src_weights fibersets)). For each `master_product_index`, the program creates the `n_sim` file structure (`sims/<sim index>/n_sims/<n_sim index>/data/inputs/` and `sims/<sim index>/n_sims/<n_sim index>/data/outputs/`). Corresponding to the `n_sim's master_product_index`, files are copied into the `n_sim` directory for a "reduced" ***Sim***, stimulation waveform, and fiber potentials. Additionally, the program writes a HOC file (i.e., `"launch.hoc"`) containing parameters for and a call to our `Wrapper.hoc` file using the Python `HocWriter` class.

To conveniently submit the `n\_sim` directories to a computer cluster, we created methods within Simulation named `export_n_sims()`, `export_run()`, and `export_neuron_files()`. The method `export_n_sims()` copies `n_sims` from our native hierarchical file structure to a target directory as defined in the system `env.json` config file by the value for the `"ASCENT_NSIM_EXPORT_PATH"` key. Within the target directory, a directory named `n_sims/` contains all `n_sims`. Each `n_sim` is renamed corresponding to its sample, model, sim, and `master_product_index` (`<sample_index>_<model_index>_<sim_index>_<master_product_index>`) and is therefore unique. Analogously, `export_run()` creates a copy of ***Run*** within the target directory in a directory named `runs/`. Lastly, `export_neuron_files()` is used to create a copy of the NEURON `*.hoc` and `*.mod` files in the target directory in directories named `"HOC_Files"` and `"MOD_Files"`, respectively.

9.1.3 Fiberset

Runner's `run()` method first loads JSON configuration files for ***Sample***, ***Model***, and ***Sim*** into memory and instantiates a Python Sample class. The Sample instance produces two-dimensional CAD files that define nerve and fascicle tissue boundaries in COMSOL from the input binary masks. The `run()` method also instantiates Python Simulation classes using the ***Model*** and ***Sim*** configurations to define the coordinates of "fibersets" where "potentials" are sampled in COMSOL to be applied extracellularly in NEURON and to define the current amplitude versus time stimulation waveform used in NEURON ("waveforms"). The Simulation class is unique in that it performs operations both before and after the program performs a handoff to Java for COMSOL operations. Before the handoff to Java, each Simulation writes `fibersets/` and `waveforms/` to file, and after the Java operations are complete, each Simulation builds folders (i.e., `n_sims/`), each containing NEURON code and input data for simulating fiber responses for a single ***Sample***, ***Model***, fiberset, waveform, and contact weighting. Each instance of the Simulation class is saved as a Python object using `Saveable` (*Python Utility Classes*, which is used for resuming operations after the `handoff()` method to Java is completed.

Within the `write_fibers()` method of the Python Simulation class, the Python Fiberset class is instantiated with an instance of the Python Sample class, ***Model***, and ***Sim***. Fiberset's `generate()` method creates a set of (x,y,z)-coordinates for each Fiberset defined in ***Sim***. The (x,y)-coordinates in the nerve cross section and z-coordinates along the length of the nerve are saved in `fibersets/`.

Fiberset's method `_generate_xy()` (first character being an underscore indicates intended for use only by the Fiberset class) defines the coordinates of simulated fibers in the cross section of the nerve according to the `"xy_parameters"` JSON Object in ***Sim*** (*Sim Parameters*). The pipeline defines (x,y)-coordinates of the fibers in the nerve cross section according to the user's selection of sampling rules (`CENTROID`, `UNIFORM_DENSITY`, `UNIFORM_COUNT`, and `WHEEL`); the pre-defined modes for defining fiber locations are easily expandable. To add a new mode for defining (x,y)-coordinates, the user must add a `"FiberXYMode"` in `src/utils/enums.py` (*Enums*) and add an `IF` statement code block in `_generate_xy()` containing the operations for constructing "points" `(List[Tuple[float]])`. The user must add the parameters to define how fibers are placed in the nerve within the `"xy_parameters"` JSON Object in ***Sim***. In ***Sim***, the user may control the "plot" parameter (Boolean) in the "fibers" JSON Object to create a figure of fiber (x,y)-coordinates on the slide. Alternatively, the user may plot a Fibserset using the `plot_fiberset.py` script (*Python Morphology Classes*).

Fiberset's private method `_generate_z()` defines the coordinates of the compartments of simulated fibers along the length of the nerve based on global parameters in `config/system/fiber_z.json` and simulation-specific parameters in the `"fibers"` JSON Object in ***Sim*** (i.e., `"mode"`, `"diameter"`, `"min"`, `"max"`, and `"offset"`).

9.1.4 Python utility classes

Enums

In the Python portions of the pipeline we use Enums which are "... a set of symbolic names (members) bound to unique, constant values. Within an enumeration, the members can be compared by identity, and the enumeration itself can be iterated over." Enums improve code readability and are useful when a parameter can only assume one value from a set of possible values.

We store our Enums in `src/utils/enums.py`. While programming in Python, Enums are used to make interfacing with our JSON parameter input and storage files easier. We recommend that as users expand upon ASCENT's functionality that they continue to use Enums, adding to existing classes or creating new classes when appropriate.

Configurable

Configurable is inherited by other Python classes in the ASCENT pipeline to grant access to parameter and data configuration JSON files loaded to memory. Configurable has built-in exceptions that it throws which are indexed negatively (-1 and below by convention) because it is intrinsically unable to inherit from Exceptionable (errors indexed +1 and above by convention), which, in turn, (Exceptionable) is configured by inheriting the Configurable class.

Configurable is an important class for developers to understand because it is the mechanism by which instances of our Python classes inherit their properties from JSON configuration files (e.g., `sample.json`, `model.json`, `sim.json`, `fiber_z.json`). The Configurable class takes three input parameters:

SetupMode (from Enums)

Either NEW or OLD which determines if Configurable loads a new JSON (from file) or uses data that has already been created in Python memory as a dictionary or list, respectively.

ConfigKey (from Enums)

The ConfigKey indicates the choice of configuration data type and is also the name of the configuration JSON file (e.g., `sample.json`, `model.json`, `sim.json`, `run.json`, `env.json`).

Config:

The Config input to Configurable can take one of three data types. If `"SetupMode"` is "OLD", the value can be a dictionary or list of already loaded configuration data. If `"SetupMode"` is "NEW", the value must be a string of the file path to the configuration file to be loaded into memory.

Example use of Configurable:

When the Sample class is instantiated in Runner, it inherits functionality from Configurable (see Sample constructor `__init__(self, exception_config: list)` in `src/core/sample.py`).

After the Sample class is instantiated, the ***Sample*** configuration (index indicated in ***Run***) is added to the Sample class with:

```
sample.add(SetupMode.OLD, Config.SAMPLE, all_configs[Config.SAMPLE.value][0])
```

With the ***Sample*** configuration available to the Sample class, the class can access the contents of the JSON dictionary. For example, in `populate()`, the Sample class gets the length of the scale bar from ***Sample*** with the following line:

`self.search(Config.SAMPLE, 'scale', 'scale_bar_length')`

Exceptionable

Exceptionable is a centralized way to organize and throw exceptions (errors) to the user's console. Exceptionable inherits functionality from Configurable. Exceptionable, like Configurable, is initialized with "SetupMode", ConfigKey, and a Config. However, the data contents for Exceptionable are specifically a list of exceptions stored in `config/system/exceptions.json`. The contents of the exceptions configuration file is a list of numbered errors with an associated text description. These contents, along with the path of the script which called exceptionable, are listed in the event of a raised exception.

Saveable

Saveable is a simple Python class that, when inherited by a Python class (e.g., *Sample* and *Simulation*) enables the class to save itself using Saveable's `save()` method. Using `pickle.dump()`, the object is saved as a Python object to file at the location of the destination path, which is an input parameter to `save()`.

9.2 Java Classes

9.2.1 ModelWrapper Class

The `ModelWrapper` class in Java takes inputs of the ASCENT_PROJECT_PATH (`env.json`, *JSON Configuration Files*) and a list of ***Run*** paths. `ModelWrapper` contains a COMSOL "model" object, model source directory (String), model destination directory (String), an `"IdentifierManager"` (*Java Utility Classes*), and `HashMaps` with key-value pairs linking COMSOL domains to unions of domains. `ModelWrapper` has accessor methods `getModel()` for retrieving the model object, `getRoot()` for retrieving the project path's root String, and `getDest()` for retrieving the default saving destination String. `ModelWrapper` also has mutator methods for changing an instance's root String (`setRoot()`) and default saving destination String (`setDest()`).

`ModelWrapper's` `main()` method starts an instance of COMSOL and loads ***Run***, ***Sample***, ***Model***, and ***Sim*** configurations as JSON Objects into memory. We developed Java class `JSONio` (*Java Utility Classes*) for reading and writing JSON Objects to file.

Since each ***Run*** contains a list of ***Model*** and ***Sim*** configurations for a single ***Sample*** (note: `n_sims/` are created for all combinations of ***Model*** and ***Sim*** for the ***Sample*** in a ***Run***), `ModelWrapper` iterates over ***Model*** configurations (e.g., different cuff electrodes or material assignments) to define the FEM geometry, mesh, assign boundary conditions and physics, and solve. The resulting FEM potentials are obtained for 1 mA applied to one of the electrode contacts while the electric potential on the other contacts is floating (i.e., condition of continuity); this is repeated for each contact to define the "bases" of the solution space [Pelot *et al.*, 2018]. For each ***Sim***, the program then creates a superposition of the "bases" for extracellular potentials at the coordinates defined in `fibersets/` and `ss_coords/` (i.e., the coordinates along the length of the nerve used to "super-sample" potentials for later creating potentials/ without the need for COMSOL). We wrote the code such that the program will continue with creating potentials/, `ss_bases/` (i.e., the potentials along the length of the nerve corresponding 1:1 to the coordinates saved in `ss_coords/`, which are added together according to the contact weighting defined by `"active_srcs"` in *Sim* to create potentials/ for specific fiber models), and NEURON simulations for any remaining ***Model*** indices even if the processes for a single ***Model*** fails. For each ***Model***, the program appends a Boolean to `"models_exit_status"` in ***Run*** (true if successful, false if not successful).

ModelWrapper.addNerve()

The `addNerve()` method adds the nerve components to the COMSOL "model" object using Java. If the `"NerveMode"` in ***Sample*** ("nerve") is "PRESENT" (*Sample Parameters*) the program creates a part instance of epineurium using the `createNervePartInstance()` method in Part (`src/model/Part.java`). The `addNerve()` method then searches through all directories in `fascicles/` for the sample being modeled, and, for each fascicle, assigns a path for the inner(s) and outer in a `HashMap`. The `HashMap` of fascicle directories is then passed to the `createNervePartInstance()` method in Part which adds fascicles to the COMSOL "model" object.

ModelWrapper.extractAllPotentials()

The user is unlikely to interface directly with ModelWrapper's `extractAllPotentials()` method in Java as it operates behind the scenes. The method takes input arguments for the project path and a run path. Using the run path, the method loads ***Run***, and constructs lists of ***Model*** and ***Sim*** for which it will call `extractPotentials()` for each fiberset. COMSOL is expecting a (3 *n*) matrix of coordinates (Double[3][n]), defining the (x,y,z)-coordinates for each of *n* points.

The Java COMSOL API methods `setInterpolationCoordinates()` and `getData()` for a model object are fast compared to the time for a machine to load a COMSOL "model" object to memory from file. Therefore, the `extractAllPotentials()` method is intentionally configured to minimize the number of times a "basis" COMSOL "model" object is loaded into memory. We accomplish this by looping in the following order: ***Model***, bases, ***Sims***, fibersets (i.e., groups of fibers with identical geometry/channels, but different (x,y)-locations and/or longitudinal offsets), then fibers. With this approach, we load each COMSOL "model" object only once (i.e., *.mph members of bases/). Within the loop, the `extractPotentials()` method constructs the bases (double[basis index][sim index][fiberset index][fiber index]) for each model (units: Volts). With the bases in memory, the program constructs the potentials for inputs to NEURON by combining bases by their contact weights and writes them to file within potentials/ (or ss_bases/), which mirrors fibersets/ (or ss_coords/) in contents.

ModelWrapper.addMaterialDefinitions()

The user is unlikely to interface directly with the `addMaterialDefinitions()` method in Java as it operates behind the scenes. The method takes an input of a list of strings containing the material functions (i.e., endoneurium, perineurium, epineurium, cuff "fill", cuff "insulator", contact "conductor", and contact "recess"), ***Model***, and a COMSOL `ModelParamGroup` for containing the material properties as a COMSOL "Parameters Group" under COMSOL's "Global Definitions". The method then loops over all material functions, creates a new material if it does not yet exist as `"mat<#>"` using Part's `defineMaterial()` method, and adds the identifier (e.g., "mat1") to the `IdentifierManager`.

ModelWrapper.addCuffPartMaterialAssignment()

The user is unlikely to interface directly with the `addCuffPartMaterialAssignment()` method in Java as it operates behind the scenes. The method loads in a cuff part primitive's labels from its IdentifierManager. The method loops over the list of material JSON Objects in the "preset" cuff configuration file. For each material function in the "preset" cuff configuration file, the method creates a COMSOL Material Link to assign a previously defined selection in a cuff part instance to a defined material.

ModelWrapper.addCuffPartMaterialAssignments()

The user is unlikely to interface directly with the `addCuffMaterialAssignments()` method in Java as it operates behind the scenes. The method loops through all part instances in the cuff configuration file, which is linked to ***Model*** by a string of its file name under the “preset” key, and calls the `addCuffMaterialAssignment()` method for each part instance. As described in `addCuffPartMaterialAssignment()` above, a material is connected to the selection within the part primitive by its label. In COMSOL, material links appear in the “Materials” node. COMSOL assigns the material links in the order of the part instances defined in the “preset” cuff configuration file, which is important since material links overwrite previous domain assignments. For this reason, it is important to list part instances in “preset” cuff files in a nested order (i.e., the outermost domains first, knowing that domains nested in space within them will overwrite earlier domain assignments).

9.2.2 Making geometries in COMSOL (Part class)

Part.createEnvironmentPartPrimitive()

The `createEnvironmentPartPrimitive()` method in Java (`src/model/Part.java`) creates a “part” within the “Geometry Parts” node of the COMSOL “model” object to generalize the cylindrical medium surrounding the nerve and electrode. Programmatically selecting domains and surfaces in COMSOL requires that geometry operations be contributed to “selections” (`csel<#>`). In this simple example of a part primitive, the `im.labels String[]` contains the string “MEDIUM” which is used to label the COMSOL selection (`csel<#>`) for the medium domain by association with an `IdentifierManager` (*Java Utility Classes*). When the geometry of the primitive is built, the resulting medium domain’s `csel<#>` can be accessed instead with the key “MEDIUM” in the `IdentifierManager`, thereby improving readability and accessibility when materials and boundary conditions are assigned in the `createEnvironmentPartInstance()` method. Furthermore, if the operations of a part primitive are modified, the indexing of the `csel<#>` labels are automatically handled.

Part.createCuffPartPrimitive()

The `createCuffPartPrimitive()` method in Java (`src/model/Part.java`) is analogous to `createEnvironmentPartPrimitive()`, except that it contains the operations required to define cuff part geometries, which are generally more complex. Examples of cuff part primitives include standard geometries for contact conductors (e.g., Ribbon Contact Primitive, Wire Contact Primitive, Circle Contact Primitive, and Rectangular Contact Primitive), cuff insulation (e.g., Tube Cuff), cuff fill (e.g., saline, mineral oil), and specific interactions of a cuff insulator and electrode contact (e.g., LivaNova-inspired helical coil) (*ASCENT Part Primitives*).

Part Instances

Part instances are a COMSOL Geometry Feature (`"pi<#>"`) in the “Geometry” node based on user-defined input parameters stored in ***Model*** and default parameters for “preset” cuffs. A part instance is an instantiation of a part primitive previously defined in the COMSOL “model” object and will take specific form based on its input parameters.

Part.createEnvironmentPartInstance()

The `createEnvironmentPartInstance()` method in Java creates a "part instance" in COMSOL's "Geometry" node based on a primitive previously defined with `createEnvironmentPartPrimitive()`. This method just applies to building the surrounding medium. The method takes inputs, with data types and examples in parentheses: `instanceID` (String: `"pi<#>"`), `instanceLabel` (String: "medium"), `mediumPrimitiveString` (String: Key for the medium part stored in the `identifierManager`), an instance of `ModelWrapper`, and ***Model*** as a JSON Object. Within the "medium" JSON Object in ***Model***, the parameters required to instantiate the environment part primitive are defined.

Part.createCuffPartInstance()

The `createCuffPartInstance()` method in Java is analogous to `createEnvironmentPartInstance()`, but it is used to instantiate cuff part geometries. We decided to separate these methods since all products of `createCuffPartInstance()` will be displaced and rotated by the same cuff shift (x,y,z) and rotation values.

Part.createNervePartInstance()

The `createNervePartInstance()` method in Part (`src/model/Part.java`) creates three-dimensional representations of the nerve sample including its endoneurium, perineurium, and epineurium. The `createNervePartInstance()` method defines domain and surface geometries and contributes them to COMSOL selections (lists of indices for domains, surfaces, boundaries, or points), which are necessary to later assign physics properties.

Fascicles

ASCENT uses CAD sectionwise files (i.e., ASCII with `.txt` extension containing column vectors for x- and y-coordinates) created by the Python Sample class to define fascicle tissue boundaries in COMSOL.

We provide the `"use_ci"` mode in ***Model*** (*Model Parameters*) to model the perineurium using COMSOL's contact impedance boundary condition for fascicles with only one inner (i.e., endoneurium) domain for each outer (i.e., perineurium) domain (*Perineurium Properties*). If `"use_ci"` mode is true, the perineurium for all fascicles with exactly one inner and one outer is represented with a contact impedance. The pipeline does not support control of using the contact impedance boundary condition on a fascicle-by-fascicle basis.

The `createNervePartInstance()` method in Part (`src/model/Part.java`) performs a directory dive on the output CAD files `samples/<sample_index>/slides/<#>/<#>/sectionwise2d/fascicles/<outer,inners>/` from the Sample class in Python to create a fascicle for each outer. Depending on the number of corresponding inners for each outer saved in the file structure and the `"use_ci"` mode in ***Model***, the program either represents the perineurium in COMSOL as a surface with contact impedance (FascicleCI: Fascicles with one inner per outer and if `"use_ci"` mode is true) or with a three-dimensional meshed domain (FascicleMesh: Fascicles with multiple inners per outer or if `"use_ci"` parameter is false).

Epineurium

The `createNervePartInstance()` method in Part (`src/model/Part.java`) contains the operations required to represent epineurium in COMSOL. The epineurium cross section is represented one of two ways:

- If `deform_ratio` in ***Sample*** is set to 1 and `"DeformationMode"` is not `"NONE"`, the nerve shape matches the `"ReshapeNerveMode"` from ***Sample*** (e.g., `"CIRCLE"`). (*Sample Parameters*). An epineurium boundary is then created from this shape.
- Otherwise, the coordinate data contained in `samples/<sample_index>/slides/<#>/<#>/sectionwise2d/nerve/0/0.txt` is used to create a epineurium boundary.

The epineurium boundary is then extruded into the third dimension. This is only performed if the `"NerveMode"` (i.e., “nerve”) in ***Sample*** is `"PRESENT"` and `n.tif` is provided (*Sample Parameters*).

Part.defineMaterial()

The user is unlikely to interface directly with the `defineMaterial()` method in Java as it operates behind the scenes to add a new material to the COMSOL “Materials” node under “Global Definitions”. The method takes inputs of the material’s identifier in COMSOL (e.g., “mat1”), function (e.g., cuff “fill”), ***Model***, a library of predefined materials (e.g., `materials.json`), a ModelWrapper instance, and the COMSOL ModelParamGroup for material conductivities. The `defineMaterial()` method uses materials present in ***Model’s*** “conductivities” JSON Object to assign to each material function in the COMSOL model (e.g., insulator, conductor, fill, endoneurium, perineurium, epineurium, or medium). The material value for a function key in ***Model*** is either a string for a pre-defined material in `materials.json`, or a JSON Object (containing unit, label, and value) for a custom material. Assigning material conductivities to material functions in this manner enables control for a user to reference a pre-defined material or explicitly link a custom material to a COMSOL model. In either the case of a predefined material in `materials.json` or custom material in ***Model***, if the material is anisotropic, the material value is assigned a string “anisotropic” which tells the program to look for independent `"sigma_x"`, `"sigma_y"`, and `"sigma_z"` values in the material JSON Object.

9.2.3 Java utility classes

IdentifierManager

In working with highly customized COMSOL FEMs, we found it convenient to abstract away from the underlying COMSOL indexing to improve code readability and enable scalability as models increase in geometric complexity. We developed a class named `IdentifierManager` that allows the user to assign their own String to identify a COMSOL geometry feature tag (e.g., `wp<#>`, `cyl<#>`, `rev<#>`, `dif<#>`) or selection (i.e., `csel<#>`).

We use `IdentifierManagers` to name and keep track of identifier labels within a `PartPrimitive`. `IdentifierManagers` assigning tags for products of individual operations to a unique “pseudonym” in HashMaps as a key-value pair. Additionally, we assign resulting selections (`"csel<#>"`) to meaningful unique pseudonyms to later assign to meshes, materials, or boundary conditions.

We keep a running total for tags of `IdentifierStates` (i.e., the total number of uses of an individual COMSOL tag) and `HashMap` of `IdentifierPseudonyms` (i.e., a `HashMap` containing key (pseudonym) and value (COMSOL tag, e.g., “wp1”)).

`IdentifierManager` has a method `next()` which takes inputs of a COMSOL tag (e.g., `wp`, `cyl`, `rev`, `dif`) or selection (i.e., `csel`) without its index and a user’s unique pseudonym String. The method appends the next index (starting at 1) to the COMSOL tag or selection and puts the COMSOL tag or selection and associated pseudonym in the `IdentifierPseudonyms` `HashMap`. The method also updates the `IdentifierStates` `HashMap` total for the additional instance of a COMSOL tag or selection.

To later reference a COMSOL tag or selection, `IdentifierManager` has a `get()` method which takes the input of the previously assigned pseudonym key and returns the COMSOL tag or selection value from the `IdentifierPseudonyms HashMap`.

To accommodate mesh recycling (see `ModelSearcher` below), we save a COMSOL model's `IdentifierManagers` to later access selections for updating model materials and physics. Therefore, we developed `IdentifierManager` methods `toJSONObject()` and `fromJSONObject()` which saves an `IdentifierManager` to a JSON file and loads an `IdentifierManager` into Java from a JSON Object, respectively.

JSONio

`JSONio` is a convenient Java class used for reading and writing JSON Objects to file. The `read()` method takes an input String containing the file path to read and returns a JSON Object to memory. The `write()` method takes an input String containing the file path for the saving destination and a JSON Object containing the data to write to file.

ModelSearcher

The `ModelSearcher` class in Java is used to look for previously created FEM meshed geometries that can be repurposed. For example, if ***Model*** configurations differ only in their material properties or boundary conditions and the previous ***Model's*** *.mph file with the mesh (i.e., `mesh.mph`) was saved, then it is redundant to build and mesh the same model geometry for a new ***Model*** configuration. The methods of the `ModelSearcher` class can save enormous amounts of computation time in parameter sweeps of ***Model*** if the mesh can be recycled. The user is unlikely to interface directly with this method as it operates behind the scenes, but if the user adds new parameter values to ***Model***, then the user must also add those values to `config/templates/mesh_dependent_model.json` to indicate whether the added parameter value needs to match between FEMs to recycle the mesh (explained further below). Generally, changes in geometry or meshing parameters need to match, but changes in material properties or boundary conditions do not, since they do not change the FEM geometry.

Specifically, this class compares ***Model*** configurations to determine if their parameters are compatible to repurpose the geometry and mesh from a previously generated COMSOL model using the `meshMatch()` method. The `meshMatch()` method takes the inputs of a reference JSON (i.e., `config/templates/mesh_dependent_model.json`, see *Mesh Dependent Model*) containing conditions for compatibility and a JSON Object for each of two ***Model*** configurations to compare. The parameter keys correspond one-to-one in ***Model*** and `mesh_dependent_model.json`. However, in `mesh_dependent_model.json`, rather than numerical or categorical values for each parameter key, the keys' values are a Boolean indicating if the values between two ***Model*** configurations must be identical to define a "mesh match". For two ***Model*** configurations to be a match, all parameters assigned with the Boolean true in `mesh_dependent_model.json` must be identical. ***Model*** configurations that differ only in values for parameters that are assigned the Boolean false are considered a mesh match and do not require that the geometry be re-meshed.

In the class's `searchMeshMatch()` method, the program looks through all ***Model*** configurations under a given ***Sample*** and applies the `meshMatch()` method. If a ***Model*** match is found, `searchMeshMatch` returns a Match class, which is analogous to the `ModelWrapper` class, using the path of the matching ***Model*** with the `fromMeshPath()` method.

9.3 NEURON Files

9.3.1 NEURON Wrapper.hoc

The `Wrapper.hoc` file coordinates all program operations to create a biophysically realistic discrete cable fiber model, simulate the fiber's response to extracellular and intracellular stimulation, and record the response of the fiber. For each fiber simulated in NEURON, outputs are saved to `<n_sim_index>/data/outputs/`. For simulations running an activation or block threshold protocol, data outputs include threshold current amplitudes. For simulation of fiber response to set amplitudes, the user may save state variables at each compartment in NEURON to file at discrete times and/or locations.

Create fiber model

Based on the flag for "fiber_type" set in `launch.hoc` (associated by a fiber type parameter in `fiber_z.json` and `FiberGeometryMode` (*Sim Parameters*)), `Wrapper.hoc` loads the corresponding template for defining fiber geometry discretization, i.e., `"GeometryBuilder.hoc"` for myelinated fibers and `"cFiberBuilder.hoc"` for unmyelinated fibers. For all fiber types, the segments created and connected in NEURON have lengths that correspond to the coordinates of the input potentials.

Intracellular stimulus

For simulations of block threshold, an intracellular test pulse is delivered at one end of the fiber to test if the cuff electrode (i.e., placed between the intracellular stimulus and the site of detecting action potentials) is blocking action potentials (*Simulation Protocols*). The intracellular stimulation parameters are defined in ***Sim*** and are defined as parameters in NEURON within the `launch.hoc` file. The parameters in ***Sim*** control the pulse delay, pulse width, pulse repetition frequency, pulse amplitude, and node/section index of the intracellular stimulus (*Sim Parameters*). For simulating activation thresholds, the intracellular stimulation amplitude should be set to zero.

Extracellular stimulus

To simulate response of individual fibers to electrical stimulation, we use NEURON's extracellular mechanisms to apply the electric potential from COMSOL at each segment of the cable model as a time-varying signal. We load in the stimulation waveform from a `n_sim's data/inputs/` directory using the `VeTime_read()` procedure within `ExtracellularStim_Time.hoc`. The saved stimulation waveform is unscaled, meaning the maximum current magnitude at any timestep is +/-1. Analogously, we read in the potentials for the fiber being simulated from `data/inputs/` using the `VeSpace_read()` procedure within `ExtracellularStim_Space.hoc`.

Recording

The NEURON simulation code contains functionality ready to record and save to file the values of state variables at discrete spatial locations for all times and/or at discrete times for all spatial locations (i.e., nodes of Ranvier for myelinated fibers or sections for unmyelinated fibers) for applied extracellular potential, intracellular stimulation amplitude, transmembrane potential, and gating parameters using `Recording.hoc`. The recording tools are particularly useful for generating data to troubleshoot and visualize simulations.

RunSim

Our procedure `RunSim` is responsible for simulating the response of the model fiber to intracellular and extracellular stimulation. Before the simulation starts, the procedure adds action potential counters to look for a rise above a threshold transmembrane potential.

So that each fiber reaches a steady-state before the simulation starts, the `RunSim` procedure initializes the fiber by stepping through large time steps with no extracellular potential applied to each compartment. `RunSim` then loops over each time step, and, while updating the value of extracellular potential at each fiber segment, records the values of flagged state variables as necessary.

At the end of `RunSim's` loop over all time steps, if the user is searching for threshold current amplitudes, the method evaluates if the extracellular stimulation amplitude was above or below threshold, as indicated by the presence or absence of an action potential for activation and block thresholds, respectively.

FindThresh

The procedure `FindThresh` performs a binary search for activation and block thresholds (*Simulation Protocols*).

Save outputs to file

At the end of the NEURON simulation, the program saves state variables as indicated with saveflags, CPU time, and threshold values. Output files are saved to the `data/outputs/` directory within its `n_sim` folder.

9.3.2 NEURON launch.hoc

The `launch.hoc` file defines the parameters and simulation protocol for modeling fiber response to electrical stimulation in NEURON and is automatically populated based on parameters in ***Model*** and ***Sim***. The `launch.hoc` file is created by the `HocWriter` class. Parameters defined in `launch.hoc` span the categories of: environment (i.e., temperature from ***Model***), simulation time (i.e., time step, duration of simulation from ***Sim***), fiber parameters (i.e., flags for fiber geometry and channels, number of fiber nodes from ***Model***, ***Sim***, and `config/system/fiber_z.json`), intracellular stimulation (i.e., delay from start of simulation, amplitude, pulse duration, pulse repetition frequency from ***Sim***), extracellular stimulation (i.e., path to waveform file in `n_sim/` folder which is always `data/inputs/waveform.dat`), flags to define the model parameters that should be recorded (i.e., Vm(t), Gating(t), Vm(x), Gating(x) from ***Sim***), the locations at which to record the parameters (nodes of Ranvier for myelinated axons from ***Sim***), and parameters for the binary search for thresholds (i.e., activation or block protocol, initial upper and lower bounds on the stimulation amplitude for the binary search, and threshold resolution for the binary search from ***Sim***). The `launch.hoc` file loads `Wrapper.hoc` which calls all NEURON procedures. The `launch.hoc` file is created by the Python `HocWriter` class, which takes inputs of the ***Sim*** directory, `n_sim/` directory, and an exception configuration. When the `HocWriter` class is instantiated, it automatically loads the `fiber_z.json` configuration file which contains all associated flags, parameters, and rules for defining a fiber's geometry and channel mechanisms in NEURON.

CHAPTER

TEN

DATA HIERARCHY

Each execution of the ASCENT pipeline requires a ***Run*** JavaScript Object Notation (JSON) configuration file (`<run_index>.json`) that contains indices for a user-defined set of JSON files. Specifically, a JSON file is defined for each hierarchical domain of information: (1) ***Sample***: for processing segmented two-dimensional transverse cross-sectional geometry of a nerve sample, (2) ***Model*** (COMSOL parameters): for defining and solving three-dimensional FEM, including geometry of nerve, cuff, and medium, spatial discretization (i.e., mesh), materials, boundary conditions, and physics, and (3) ***Sim*** (NEURON parameters): for defining fiber models, stimulation waveforms, amplitudes, and durations, intracellular test pulses (for example, when seeking to determine block thresholds), parameters for the binary search protocol and termination criteria for thresholds, and flags to save state variables. These configurations are organized hierarchically such that ***Sample*** does not depend on ***Model*** or ***Sim***, and ***Model*** does not depend on ***Sim***; thus, changes in ***Sim*** do not require changes in ***Model*** or ***Sample***, and changes in ***Model*** do not require changes in ***Sample*** (Figure A).

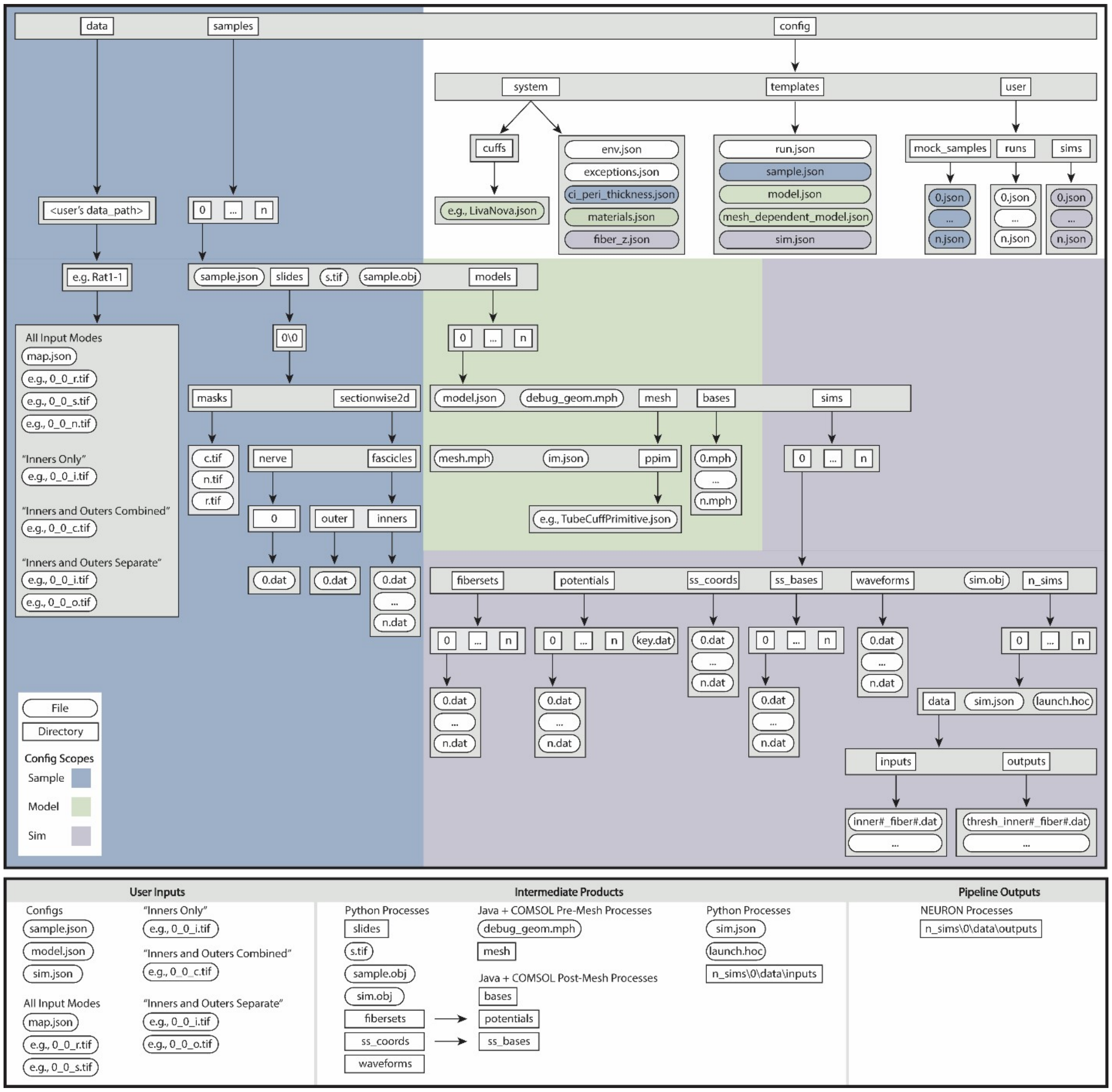


Figure A. ASCENT pipeline file structure in the context of Sample (blue), Model (green), and Sim (purple) configurations. *JSON Overview* describes the JSON configuration files an their contents, and *JSON Parameters* details the syntax and data types of the key-value parameter pairs.

10.1 Batching and sweeping of parameters

ASCENT enables the user to batch rapidly simulations to sweep cuff electrode placement on the nerve, material properties, stimulation parameters, and fiber types. The first process of ASCENT prepares ready-to-submit NEURON simulations to model response of fibers to extracellular stimulation. The second process of ASCENT uses Python to batch NEURON jobs to a personal computer or compute cluster to simulate fiber response to extracellular stimulation. Each task submitted to a CPU simulates the response of a single fiber to either a set of finite amplitudes or a binary search for threshold of activation or block, therefore creating an "embarrassingly parallel" workload.

Groups of fibers from the same ***Sample***, ***Model***, ***Sim***, waveform, contact weight (i.e., `"src_weights"` in ***Sim***), and fiberset (i.e., a group of fibers with the same geometry and channels and occupy different (x,y)-locations in the nerve cross section) are organized in the same `n_sim/` directory.

A ***Run*** creates simulations for a single ***Sample*** and all pairs of listed ***Model(s)*** and ***Sim(s)***. A user can pass a list of ***Run*** configurations in a single system call with `"python run pipeline <run_indices>"` to simulate multiple ***Sample*** configurations in the same system call.

Sample and ***Model*** cannot take lists of parameters. Rather, if the user would like to assess the impact of ranges of parameters for ***Sample*** or ***Model***, they must create additional ***Sample*** and ***Model*** configuration files for each parameter value.

Sim can contain lists of parameters in `"active_srcs"` (i.e., cuff electrode contact weightings), `"fibers"`, `"waveform"`, and `"supersampled_bases"`.

CHAPTER

ELEVEN

PUBLICATIONS UTILIZING ASCENT

Coming soon…

CHAPTER

TWELVE

REFERENCES

CHAPTER

THIRTEEN

ASCENT VALIDATION

13.1 Sim4Life validation

We designed test simulations to verify ASCENT's activation thresholds. The verifications were performed by The Foundation for Research Technologies in Society (IT'IS) with the Sim4Life (Zurich, Switzerland) simulation platform. Running the following simulations required modification of the Sim4Life solver to implement the required electrical anisotropy of tissue conductivities and the boundary condition to represent the thin layer approximation used to model the perineurium.

13.1.1 Monofascicular rat nerve model

We validated activation thresholds for fibers seeded in a model of a rat cervical vagus nerve instrumented with a bipolar cuff electrode (Figure A and B).

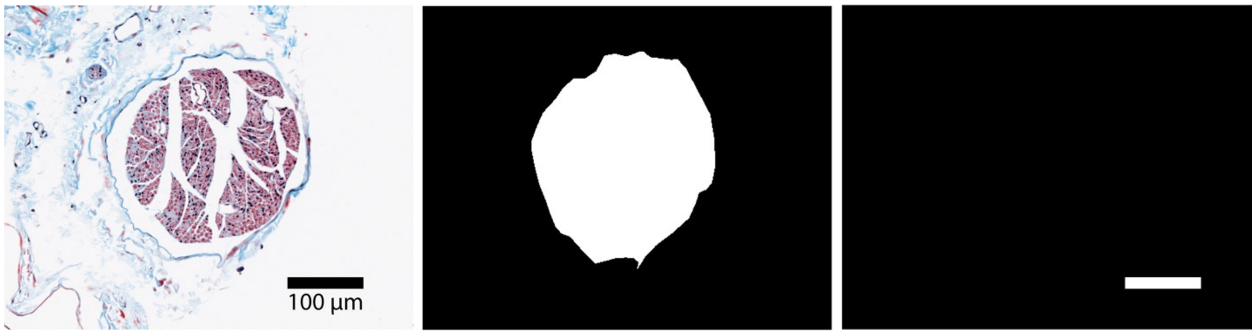


Figure A. Raw histology image (r.tif), segmented histology (i.tif), and scalebar (s.tif) of a rat cervical vagus nerve sample that served as inputs to define the cross section of the nerve in the FEM.

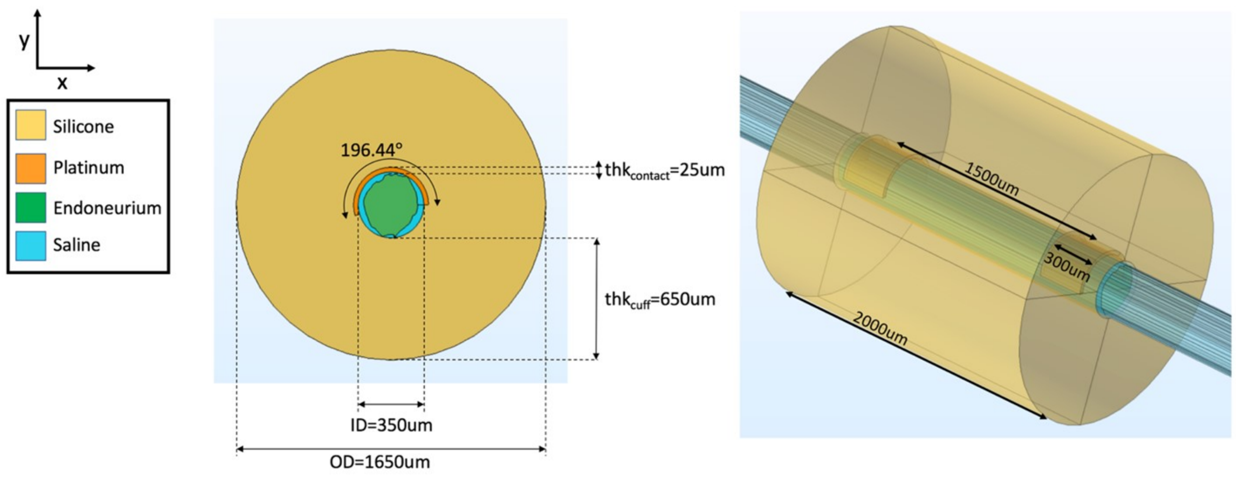


Figure B. FEM of a rat cervical vagus nerve sample instrumented with a bipolar cuff electrode.

The conductivity values applied to the rat cervical FEM are provided in Table A, and the boundary conditions applied are provided in Table B.

Table A. Conductivity values for FEM of rat cervical vagus nerve. These values were also used in multifascicular nerve model and human model verifications.

Table B. Boundary conditions used in FEM of rat cervical vagus nerve, multifascicular dummy nerve, and human cervical vagus nerve.

We compared thresholds for 100 5.7 µm myelinated axons (MRG model) seeded in the cross section of the nerve in response to a single 100 µs duration monophasic rectangular pulse. The differences in thresholds between ASCENT and IT'IS model implementations was <4.2% for all fibers, demonstrating strong agreement (Figure C).

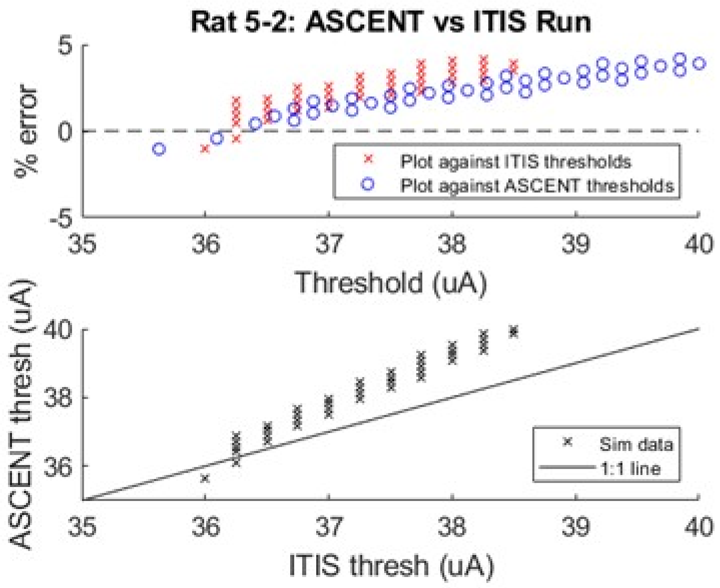


Figure C. Comparison of activation thresholds for the rat cervical vagus nerve implementation in ASCENT and Sim4Life.

13.1.2 Multifascicular dummy nerve model

We validated activation thresholds for fibers seeded in a multifascicular dummy nerve instrumented with a bipolar cuff electrode (Figure D and E). The segmented histology was created using our `mock_morphology_generator.py` script (*Mock Morphology*).

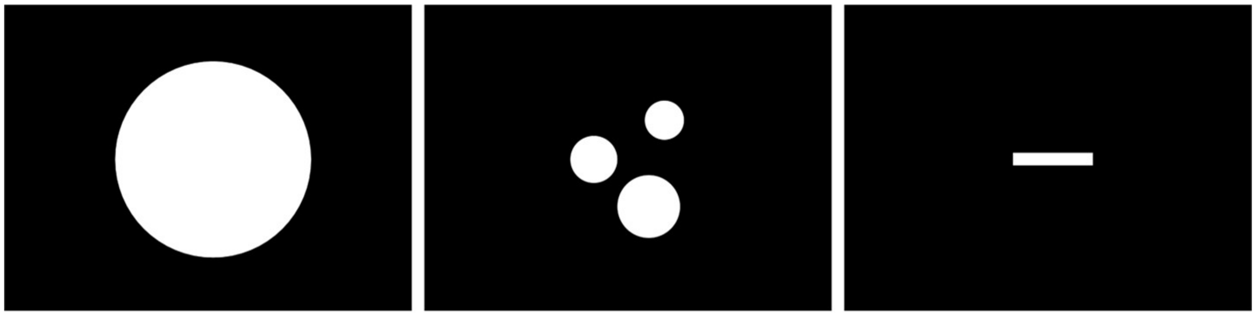


Figure D. Mock morphology inputs to the define tissue boundaries for a multifascicular dummy nerve for validation with Sim4Life. Scale bar is 100 µm long. The nerve is a perfect circle (diameter = 250 µm, centered at (x,y)=(0,0) µm). The inners are also perfect circles: (1) diameter = 50 µm, centered at (x,y)=(40,50) µm, (2) diameter = 60 µm, centered at (x,y)=(-50,0) µm, and (3) diameter = 80 µm, centered at (x,y)=(20,-60) µm.

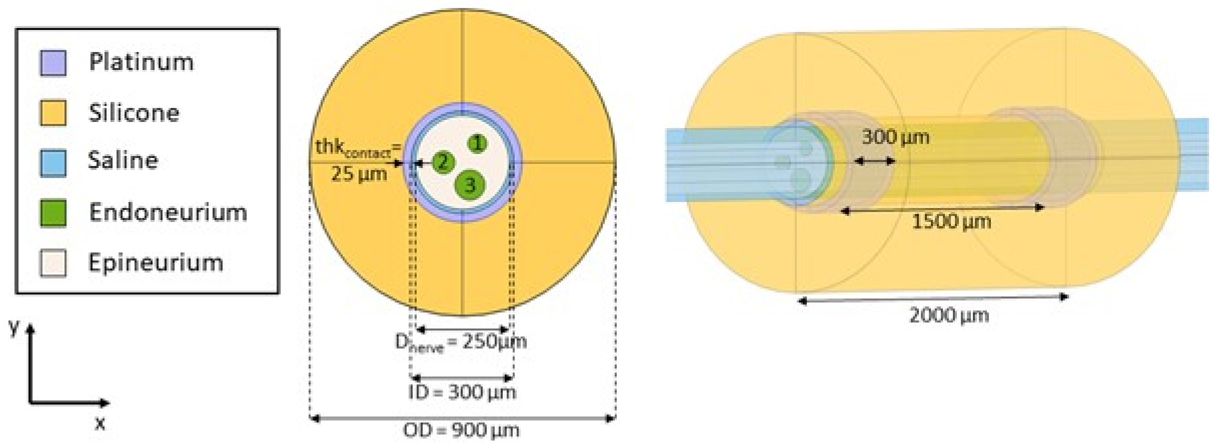


Figure E. FEM of a multifascicular nerve sample instrumented with a bipolar cuff electrode.

The conductivity values applied to multifascicular nerve sample finite element model are provided in Table A, and the boundary conditions applied are provided in Table B.

We seeded a single 5.7 µm diameter fiber in the center of each fascicle. Between the ASCENT and IT'IS implementations, there was less than a 3% difference in threshold to a single 100 µs duration monophasic rectangular pulse.

13.1.3 Multifascicular human nerve model

We validated activation thresholds for fibers seeded in a multifascicular human cervical vagus nerve instrumented with a LivaNova bipolar cuff electrode (Figure F and G). The segmented histology was created using Nikon NIS-Elements.

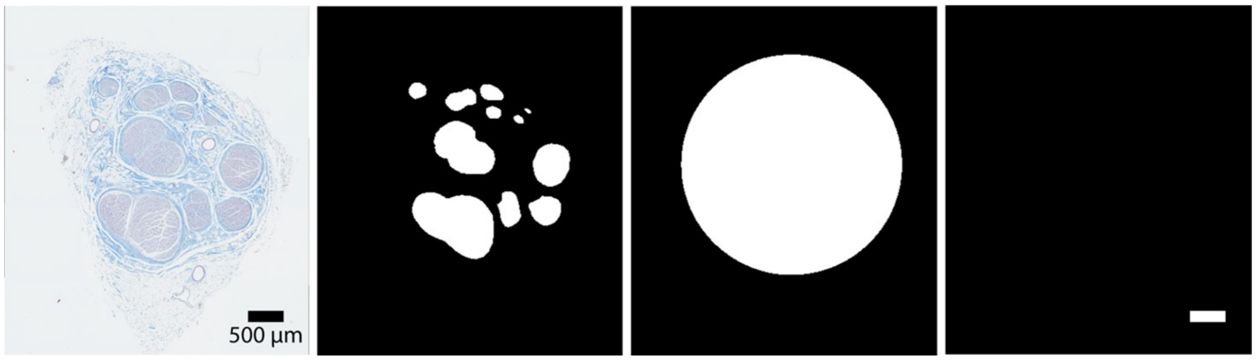


Figure F. Raw histology image (r.tif), segmented inners (i.tif), segmented nerve (n.tif), and scale bar (s.tif) of a human cervical vagus nerve sample that served as inputs to define the cross section of the nerve in the FEM for validation with Sim4Life.

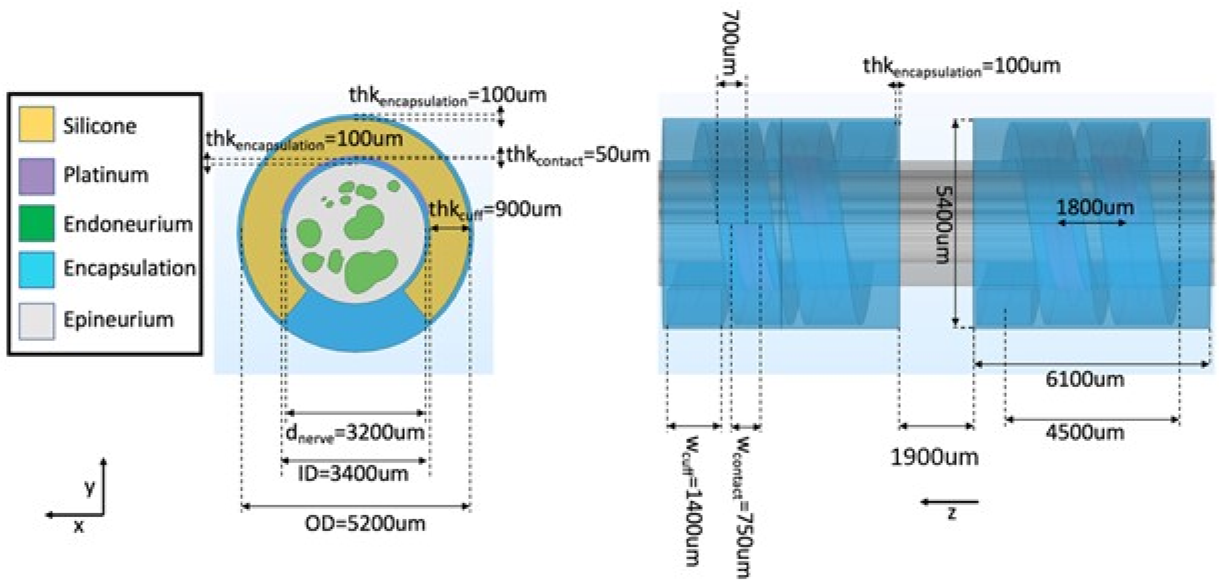


Figure G. FEM of a human cervical vagus nerve sample instrumented with a LivaNova cuff electrode.

The conductivity values applied to the human cervical vagus nerve sample finite element model are provided in Table A, and the boundary conditions applied are provided in Table B.

We seeded 5.7 µm diameter fibers in each fascicle. Between the ASCENT and IT'IS implementations, there was less than 2.5% difference to a single 100 µs duration monophasic rectangular pulse.

13.2 Comparison of MRG fit to Bucksot 2019

13.2.1 Comparison of MRG fit to Bucksot et al. 2019

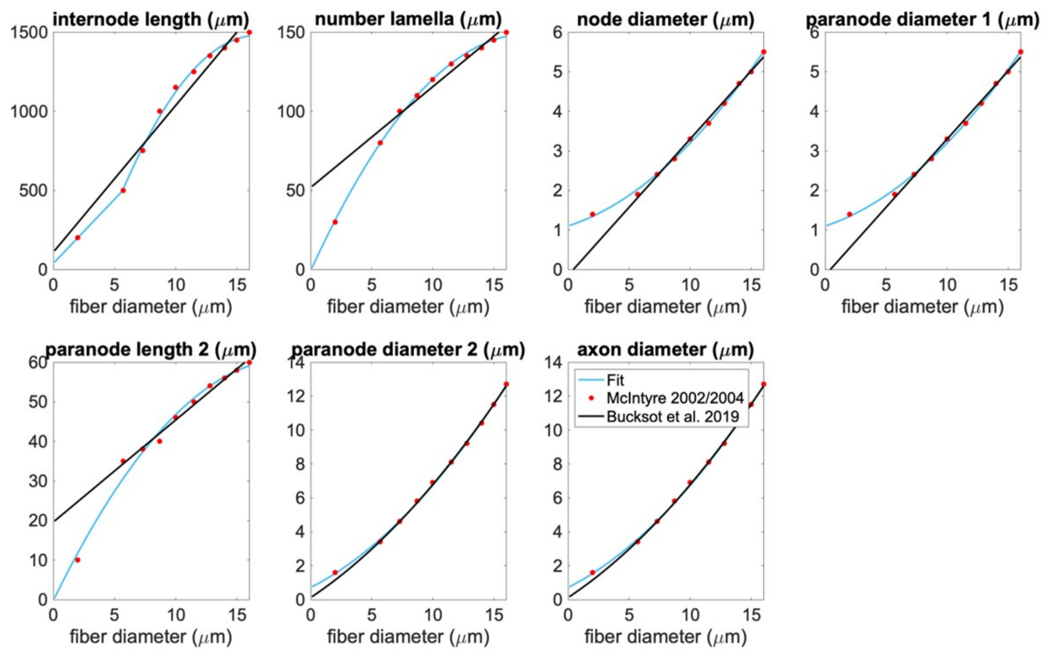
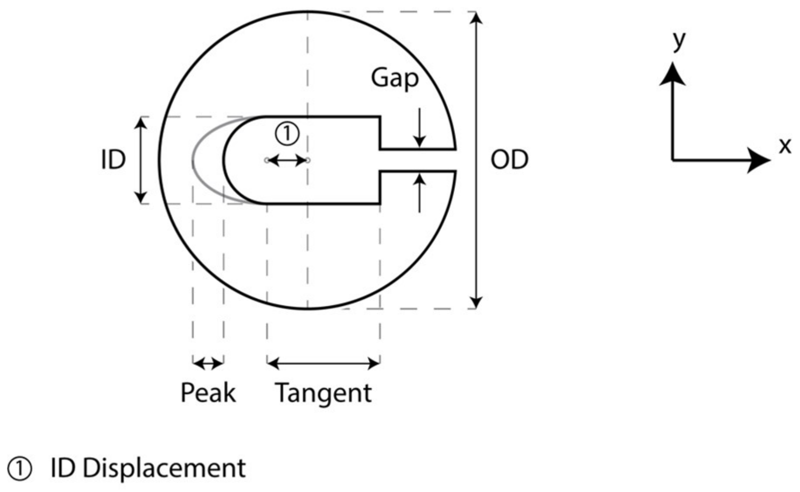


Figure A. Our piecewise polynomial fits to published MRG fiber parameters compared to the Bucksot et al. 2019's interpolation [Bucksot *et al.*, 2019]. Single quadratic fits were used for all parameters except for internode length, which has a linear fit below 5.643 µm (using MRG data at 2 and 5.7 µm) and a single quadratic fit at diameters greater than or equal to 5.643 µm (using MRG data >= 5.7 µm); 5.643 µm is the fiber diameter at which the linear and quadratic fits intersected. The fiber diameter is the diameter of the myelin. "Paranode 1" is the MYSA section, "paranode 2" is the FLUT section, and "internode" is the STIN section. The axon diameter is the same for the node of Ranvier and MYSA ("node diameter"), as well as for the FLUT and STIN ("axon diameter"). The node and MYSA lengths are fixed at 1 and 3 m, respectively, for all fiber diameters.

13.3 Micro Leads cuff measurements

We collected and measured images of 200, 300, and 400 µm Micro-Leads Neuro cuffs (Somerville, MA) (Figure A and Table A).



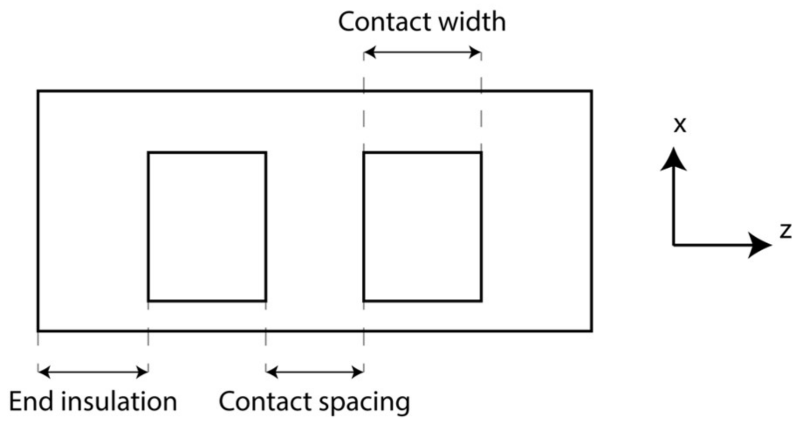


Figure A. Micro-Leads cuff measurements taken for 200, 300, and 400 µm inner diameter cuffs. Inner diameter (ID) displacement is defined from the center of the circle of the outer diameter (OD) to the center of the semi-circle for the inner diameter.

Table A. Cuff measurements (units: micrometer) for 200, 300, and 400 µm inner diameter Micro-Leads cuffs.

We also collected and measured images of 100 µm Micro-Leads cuffs, which had a different cross section from the larger diameter cuffs and are therefore reported separately (Figure B and Table B).

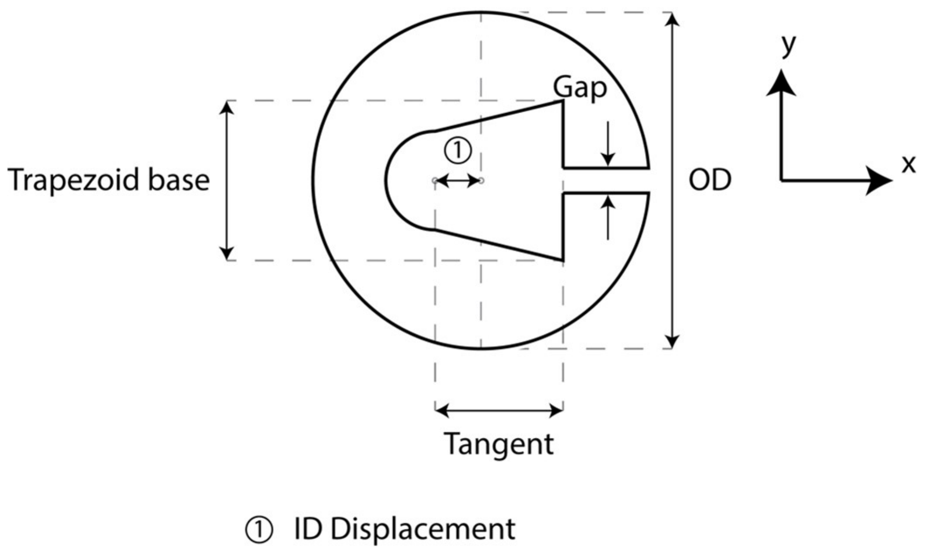


Figure B. Micro-Leads cuff measurements taken for 100 μm inner diameter cuffs. Inner diameter (ID) displacement is defined from the center of the circle of the outer diameter (OD) to the center of the semi-circle for the inner diameter.

Table B. Cuff measurements (units: micrometer) for 100 μm inner diameter Micro-Leads cuffs.

For all "preset" cuff JSON files based on our Micro-Leads cuff measurements, we recessed the contacts by 50 μm. Though we were not able to directly measure the recess depth of the contacts, Micro-Leads informed us that the surface of the electrode metal could never be recessed by more than ~60-70 μm.

CHAPTER

FOURTEEN

CHANGELOG

14.1 ASCENT v1.1.3

Released on 2022-07-26 - GitHub

14.2 ASCENT v1.1.2

Released on 2022-05-10 - GitHub

14.3 ASCENT v1.1.1

Released on 2022-01-27 - GitHub

14.4 ASCENT v1.1.0

Released on 2021-11-15 - GitHub

14.5 ASCENT v1.0.3

Released on 2021-09-10 - GitHub

14.6 ASCENT v1.0.2

Released on 2021-09-01 - GitHub

14.7 ASCENT v1.0.1

Released on 2021-08-05 - GitHub

14.8 ASCENT v1.0.0

Released on 2021-07-25 - GitHub

BIBLIOGRAPHY

[Blo07] Victor Blomqvist. Pymunk: a easy-to-use pythonic rigid body 2d physics library (version 6.0.0). 2007. URL: https://www.pymunk.org.

[BK79] I A Boyd and K U Kalu. Scaling factor relating conduction velocity and diameter for myelinated afferent nerve fibres in the cat hind limb. *The Journal of Physiology*, 289(1):277–297, April 1979. URL: https://doi.org/10.1113/jphysiol.1979.sp012737, doi:10.1113/jphysiol.1979.sp012737.

[Bra00] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[BWR+19] Jesse E. Bucksot, Andrew J. Wells, Kimiya C. Rahebi, Vishnoukumaar Sivaji, Mario Romero-Ortega, Michael P. Kilgard, Robert L. Rennaker, and Seth A. Hays. Flat electrode contacts for vagus nerve stimulation. *PLOS ONE*, 14(11):e0215191, November 2019. URL: https://doi.org/10.1371/journal.pone.0215191, doi:10.1371/journal.pone.0215191.

[CR11] William D Callister and David G Rethwisch. *Fundamentals of materials science and engineering an integrated approach 4E*. John Wiley & Sons, Nashville, TN, December 2011.

[C+--] Maxime Chalton and others. Pyclipper. 2015–. URL: https://github.com/fonttools/pyclipper.

[Cla15] Alex Clark. Pillow (pil fork) documentation. 2015. URL: https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf.

[dAlcantaraSF08] Adriana Cristina Licursi de Alcântara, Helio Cesar Salgado, and Valéria Paula Sassoli Fazan. Morphology and morphometry of the vagus nerve in male and female spontaneously hypertensive rats. *Brain Research*, 1197:170–180, March 2008. URL: https://doi.org/10.1016/j.brainres.2007.12.045, doi:10.1016/j.brainres.2007.12.045.

[dP97] Michael de Podesta. *Understanding the properties of matter*. Routledge, London, England, January 1997.

[GC--] Eric Gazoni and Charlie Clark. Openpyxl - a python library to read/write excel 2010 xlsx/xlsm files. 2020–. URL: https://openpyxl.readthedocs.io.

[GB67] L. A. Geddes and L. E. Baker. The specific resistance of biological material—a compendium of data for the biomedical engineer and physiologist. *Medical &amp$\mathsemicolon $ Biological Engineering*, 5(3):271–293, May 1967. URL: https://doi.org/10.1007/bf02474537, doi:10.1007/bf02474537.

[GJB84] F. L. H. Gielen, W. Wallinga-de Jonge, and K. L. Boon. Electrical conductivity of skeletal muscle tissue: experimental results from different musclesin vivo. *Medical &amp$\mathsemicolon $ Biological Engineering &amp$\mathsemicolon $ Computing*, 22(6):569–577, November 1984. URL: https://doi.org/10.1007/bf02443872, doi:10.1007/bf02443872.

[G+--] Sean Gillies and others. Shapely: manipulation and analysis of geometric objects. 2007–. URL: https://github.com/Toblerity/Shapely.

[GM94] Warren M. Grill and J. Thomas Mortimer. Electrical properties of implant encapsulation tissue. *Annals of Biomedical Engineering*, 22(1):23–33, January 1994. URL: https://doi.org/10.1007/bf02368219, doi:10.1007/bf02368219.

[GSTG08] Yanina Grinberg, Matthew A. Schiefer, Dustin J. Tyler, and Kenneth J. Gustafson. Fascicular perineurium thickness, size, and position affect model predictions of neural excitation. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 16(6):572–581, December 2008. URL: https://doi.org/10.1109/tnsre.2008.2010348, doi:10.1109/tnsre.2008.2010348.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/s41586-020-2649-2.

[HK17] Kenneth Horch and Daryl Kipke. *Neuroprosthetics: Theory And Practice*. Series On Bioengineering And Biomedical Engineering. WS Professional, Singapore, Singapore, 2 edition, May 2017.

[Hun07] J. D. Hunter. Matplotlib: a 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[JrB65] James B. Ranck Jr. and Spencer L. BeMent. The specific impedance of the dorsal columns of cat: an anisotropic medium. *Experimental Neurology*, 11(4):451–463, April 1965. URL: https://doi.org/10.1016/0014-4886(65)90059-2, doi:10.1016/0014-4886(65)90059-2.

[Kun--] Ken Kundert. Quantiphy. 2016–. URL: https://pypi.org/project/quantiphy/.

[MGST04] Cameron C. McIntyre, Warren M. Grill, David L. Sherman, and Nitish V. Thakor. Cellular effects of deep brain stimulation: model-based analysis of activation and inhibition. *Journal of Neurophysiology*, 91(4):1457–1469, April 2004. URL: https://doi.org/10.1152/jn.00989.2003, doi:10.1152/jn.00989.2003.

[MRG02] Cameron C. McIntyre, Andrew G. Richardson, and Warren M. Grill. Modeling the excitability of mammalian nerve fibers: influence of afterpotentials on the recovery cycle. *Journal of Neurophysiology*, 87(2):995–1006, February 2002. URL: https://doi.org/10.1152/jn.00353.2001, doi:10.1152/jn.00353.2001.

[McN--] John McNamara. Xlsxwriter. 2017–. URL: https://xlsxwriter.readthedocs.io.

[pdt20] The pandas development team. Pandas-dev/pandas: pandas. February 2020. URL: https://doi.org/10.5281/zenodo.3509134, doi:10.5281/zenodo.3509134.

[PBG17] N A Pelot, C E Behrend, and W M Grill. Modeling the response of small myelinated axons in a compound nerve to kilohertz frequency signals. *Journal of Neural Engineering*, 14(4):046022, June 2017. URL: https://doi.org/10.1088/1741-2552/aa6a5f, doi:10.1088/1741-2552/aa6a5f.

[PBG18] Nicole A Pelot, Christina E Behrend, and Warren M Grill. On the parameters used in finite element modeling of compound peripheral nerves. *Journal of Neural Engineering*, 16(1):016007, December 2018. URL: https://doi.org/10.1088/1741-2552/aaeb0c, doi:10.1088/1741-2552/aaeb0c.

[PGC+20] Nicole A. Pelot, Gabriel B. Goldhagen, Jake E. Cariello, Eric D. Musselman, Kara A. Clissold, J. Ashley Ezzell, and Warren M. Grill. Quantified morphology of the cervical and subdiaphragmatic vagus nerves of human, pig, and rat. *Frontiers in Neuroscience*, November 2020. URL: https://doi.org/10.3389/fnins.2020.601479, doi:10.3389/fnins.2020.601479.

[PTG18] Nicole A. Pelot, Brandon J. Thio, and Warren M. Grill. Modeling current sources for neural stimulation in COMSOL. *Frontiers in Computational Neuroscience*, June 2018. URL: https://doi.org/10.3389/fncom.2018.00040, doi:10.3389/fncom.2018.00040.

[RA93] F. Rattay and M. Aberham. Modeling axon membranes for functional electrical stimulation. *IEEE Transactions on Biomedical Engineering*, 40(12):1201–1209, 1993. URL: https://doi.org/10.1109/10.250575, doi:10.1109/10.250575.

[Shi--] Pete Shinners. Pygame. 2011–. URL: https://www.pygame.org.

[Sto95] C. Stolinski. Structure and composition of the outer connective tissue sheaths of peripheral nerve. *Journal of Anatomy*, 186(Pt 1):123, February 1995. Publisher: Wiley-Blackwell. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1167278/ (visited on 2022-06-06).

[SGJ15] Danielle Sundt, Nikita Gamper, and David B. Jaffe. Spike propagation through the dorsal root ganglia in an unmyelinated sensory neuron: a modeling study. *Journal of Neurophysiology*, 114(6):3140–3153, December 2015. URL: https://doi.org/10.1152/jn.00226.2015, doi:10.1152/jn.00226.2015.

[TPO+14] Jenny Tigerholm, Marcus E. Petersson, Otilia Obreja, Angelika Lampert, Richard Carr, Martin Schmelz, and Erik Fransén. Modeling activity-dependent changes of axonal spike conduction in primary afferent c-nociceptors. *Journal of Neurophysiology*, 111(9):1721–1735, May 2014. URL: https://doi.org/10.1152/jn.00777.2012, doi:10.1152/jn.00777.2012.

[Tom06] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[VdWSchonbergerNI+14] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. Scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[VGO+20] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert-Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vin\'ıcius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodr\'ıguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, and Yoshiki Vázquez-Baeza and. SciPy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, February 2020. URL: https://doi.org/10.1038/s41592-019-0686-2, doi:10.1038/s41592-019-0686-2.

[Was21] Michael L. Waskom. Seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021. URL: https://doi.org/10.21105/joss.03021, doi:10.21105/joss.03021.

[WSRT84] A. Weerasuriya, R.A. Spangler, S.I. Rapoport, and R.E. Taylor. AC impedance of the perineurium of the frog sciatic nerve. *Biophysical Journal*, 46(2):167–174, August 1984. URL: https://doi.org/10.1016/s0006-3495(84)84009-6, doi:10.1016/s0006-3495(84)84009-6.

[WDA+16] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole

Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, March 2016. URL: https://doi.org/10.1038/sdata.2016.18, doi:10.1038/sdata.2016.18.

[WesMcKinney10] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, 56 – 61. 2010. doi:10.25080/Majora-92bf1922-00a.