CONVERTING A NEURON-MORPHOLOGY RECONSTRUCTION SYSTEM: OPEN-SCIENCE DESIGN AND IMPLEMENTATION

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Zakariyya Mughal

May 2016

CONVERTING A NEURON-MORPHOLOGY RECONSTRUCTION SYSTEM: OPEN-SCIENCE DESIGN AND IMPLEMENTATION

Zakariyya Mughal

APPROVED:

Acknowledgements

Firstly, I am grateful for working with for the guidance and mentorship of my advisor Dr. Ioannis A. Kakadiaris — in both research and personal matters. Dr. Kakadiaris kept me on track and made sure that I had the resources to keep organized. His many suggestions for improving my work were valuable and those lessons will stay with me.

In addition, I would like to thank my committee, Prof. Shishir Shah and Prof. Emanuel Papadakis for asking interesting questions and pushing me to learn and try new ideas both inside and outside the classroom.

I would like to also thank Prof. Demetrio Labate, Prof. Emanuel Papadakis, Pankaj Singh, Burcin Ozcan, Pooran Negi, and Paul Hernandez-Herrera for enlightening discussions during our neuroscience research meetings and the image analysis seminars in the Department of Mathematics.

To the members of the Computational Biomedicine Lab over the years of my undergraduate and graduate studies — I learned a lot from our conversations which sparked ideas for further research and new ways of approaching computer science and software engineering. Being able to learn from the diversity of projects in the lab was a quite enjoyable experience.

I would like to acknowledge my family for their patience during the many hours that I would dissappear to work at the lab. I could not have completed my thesis without their help and encouragement.

CONVERTING A NEURON-MORPHOLOGY RECONSTRUCTION SYSTEM: OPEN-SCIENCE DESIGN AND IMPLEMENTATION

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Zakariyya Mughal

May 2016

Abstract

The thesis describes the conversion of the Online Reconstruction and functional Imaging Of Neurons (ORION) system for neuron-morphology reconstruction from an interpreted language to a compiled language. The motivation of this conversion is to provide a tool that can be used by neuroscience researchers to analyze their own neuron data and compare the output against both manual and automated tracings. This is in line with the goals of open science: a movement that seeks to make the findings and processes of research more widely available for peer review and reproducibility. By collaboratively sharing both neuron-imaging data and code between organizations, it is possible to compare the results of multiple methods without reimplementing all the stages of the reconstruction pipeline.

In order to release the existing algorithm so that it can easily be incorporated into other tools, the implementation must be rewritten in a different language. This presents a challenge because the languages have vastly different paradigms. As a result, much of the existing code needs to be analyzed to determine any changes needed to the design. Creating a new implementation also means that the new system can be designed with modifiability in mind so that future changes can be easily incorporated. The specific objectives are to (i) analyze the ORION algorithm and implementation to determine the architecture for the new system that is efficient and extensible; (ii) integrate the system into a popular toolkit for biomedical image analysis for ease-of-use and visualization; (iii) develop a test suite of both the individual components (unit testing) and across the whole system (integration tests); and (iv) ensure that the software gives reproducible results by making it easy to build and distribute.

The reconstruction of neuron morphology from microscopy imaging data is an invaluable method for understanding neuron characteristics. However, due to the cost in time and effort, manual neuron reconstruction is not feasible for large-scale analysis of neuron datasets. This implementation provides a working method for determining neuron morphology that can be used to collect statistical properties from various neuron data that can also be extended by the community.

Keywords: neurons, cell morphology, biomedical image analysis, software engineering

Contents

List of Figures

List of Algorithms

Acronyms

ABI : Application Binary Interface

API : Application Programming Interface

DIADEM : Digital Reconstruction of Axonal and Dendritic Morphology

EEG : Electroencephalography

FFT : Fast-Fourier transform

fMRI : Functional Magnetic Resonance Imaging

NIF : Neuroscience Information Framework

NITRC : Neuroimaging Informatics Tools and Resources Clearinghouse

ORION : Online Reconstruction and functional Imaging Of Neurons

SDLC : Systems Development Life Cycle

SVM : Support Vector Machine

TDD : Test-Driven Development

Notation

$\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$: Convolution of x and y

$\boldsymbol{\mathcal{F}}\left\{\cdot\right\}$: Fourier transform of argument

$\boldsymbol{\mathcal{F}^{-1}}\left\{\cdot\right\}$: Inverse Fourier transform of argument

$\widehat{\boldsymbol{x}}$: Frequency domain representation of signal x such that $\widehat{x} = \mathcal{F}\left\{x\right\}$

$\boldsymbol{z} = \boldsymbol{x} \odot \boldsymbol{y}$: Hadamard product (or pointwise/entrywise product) of $\overrightarrow{x}$ and $\overrightarrow{y}$ such that $\overrightarrow{z}_i = \overrightarrow{x}_i \overrightarrow{y}_i$

ORION3 : Version 3 of the ORION algorithm described in [1]

ORION$^{\mathbf{c}}$: C version of ORION3

ORION$^{\mathbf{m}}$: MATLAB version of ORION3

ORION$^{\mathbf{m \to c}}$: Test setup that uses the output of previous stages of ORION$^{\mathrm{m}}$ as input for a stage of ORION$^{\mathrm{c}}$

“Each discovery made by an investigator in a basic research laboratory has much larger implications today. The sum of the work in basic biology represents a rapidly expanding tool kit for engineers and inventors to use to construct items of value to society.”

David Baltimore in *How Biology Became an Information Science*, 2001 [2]

1 Introduction

This thesis aims to detail the design and reimplementation of a neuron-morphology reconstruction system. This system is the result of converting the existing ORION3 [1] system from MATLAB [3] code to native C/C++ code. This conversion process requires an analysis of the existing code to understand its structure and create a plan for replicating the same functionality in the in the new system.

This software project can be categorized as a *rewrite*; that is, a replication of an existing software system without reusing the existing code. In software engineering, the consensus on rewriting software from scratch is that it is difficult and that teams should avoid rewrites [4]. This view arises because there are several challenges and risks associated with rewriting large systems. Rewrites often take a long time and instead of adding features, development time is spent on redesign and reimplementation of old features. In addition, all the institutional knowledge that came from years of bug fixes is often lost with a rewrite. Rewrites are often expensive in terms of time and effort and rarely pay off as much as product owners wish. Therefore, it is preferable to work on slowly refactoring the code rather than a complete rewrite. Refactoring is a process where instead of throwing away the existing system, the development proceeds by making small, incremental changes over an extended period in order to avoid getting the software into a broken state, while steadily improving the maintainability and reusability of the software.

1.1 Motivation

In order to understand why this project is being undertaken, it is important to understand why a rewrite is necessary as opposed to refactoring the existing codebase. Both options require an analysis of the project outcomes and deliverables, that is, a concrete set of goals that will direct the project development.

These goals can be classified as either scientific outcomes or engineering deliverables. Scientific outcomes are motivated by goals that advance the state of scientific knowledge while engineering deliverables focus on specific technical aspects of the project that make future project maintenance and growth possible. This section will only cover the scientific outcomes while engineering deliverables are covered in Section 1.3.

The *primary scientific outcome* of this thesis is a system for neuron reconstruction. This system is designed with the principles of open science in mind so that it is usable by biologists to study neuroanatomy. Section 1.1.2 contains further discussion on open-science and the challenge of reproducibility that needs to be addressed by computational science projects such as this when trying to achieve the goal of open-science.

The *secondary scientific outcome* is to make this system compatible with existing tools for image analysis so that it can be compared against other methods as part of the BigNeuron project. The role of BigNeuron in neuroscience research is covered in Section 1.1.4.

The following sections contain an overview of the context of this project first within the general context of scientific software (Section 1.1.1) and finally narrow down its role relative to the specific context of neuron reconstruction (Section 1.5).

1.1.1 Scientific software

Quantitative methods are an essential part of scientific research. The experimental sciences depend on the dissemination of the methods used for each study so that it is clear how results are obtained and analyzed. With the rapid increase in computing power, storage, and availability, it has become easier to collect and process larger and more complex datasets using sophisticated methods and this has made software and software development an important part of all experimental fields [5, 6]. To produce reproducible results, some common approaches to this change are to publish either a description of the algorithm or refer to software that can be obtained separately (either in binary or source code form).

Despite these approaches, there are still several challenges to successfully reproducing a published method. A textual description of an algorithm is rarely a complete description of how an algorithm is implemented. Sophisticated methods often have tiny details that are mistakenly left out which may be essential for the rest of the processing. One specific area where this occurs often is in data preprocessing and annotation stages which can involve human interaction; as a result of this interaction, some assumptions may not be written down. For example, if the data contain instances with missing values, these instances may be removed based on some criteria. If these criteria are not clearly written down, it can be difficult to apply the same procedure again. This is why many statisticians recommend that the raw data and the tidy data should both be available and if possible, manual processing should be avoided in the data preprocessing step so that it is clear how to regenerate the tidy data from the raw data [7–10]. Referencing readily available software packages gives other researchers direct access to the original method used. However, even here, there can be problems. Even before running the software, it is important to ensure that the software is available years later. This means not only the software itself, but all its dependencies. This can quickly become complicated as technology advances: changes in the Application Programming Interface (API) or Application Binary Interface (ABI) can cause incompatibility issues for both software distributed in source form or binary form. Both source code and binary software can have dependencies on platforms (e.g., operating systems, runtime systems, and computer architecture) and licensing of components that can hinder others from using the software.

Even more precarious is when a dependency used by the software no longer behaves the same way as it did in previous versions. This can lead to software that appears to run, but gives unintended results. Maintaining backwards compatibility for software is difficult since there are many parts to a non-trivial software system. It may be possible to identify these compatibility problems using testing (for more discussion, see Chapter 5). Resolving the exact versions of libraries and toolchains needed to build and run can be frustrating and is commonly known as *dependency hell* [11–13]. This problem can become daunting when dealing with multiple platforms and many libraries. As newer versions of these libraries are released, the maintenance phase of the *systems-development life cycle* becomes more important (further discussion in

Section 1.2). Unmaintained software is prone to what is known as *bit rot* — that is, the process through which software that was working no longer works due to changes in the surrounding software ecosystem. There has been some work to prevent bit rot by recording a static copy of the software environment, but digital preservation (comprising both computing machinery and software) is in its infancy and has not caught up to that of paper-based materials [14–16].

1.1.2 Open science and scientific software engineering

As science becomes more oriented towards using computational tools, the ideals of reproducibility and statistical hypothesis testing become more difficult to achieve. In recent years, there has been a push by researchers to incorporate *open-science* practices in their work. One prominent advocate for open science defines this as follows:

> “Open science is the idea that scientific knowledge of all kinds should be openly shared as early as is practical in the discovery process.”
>
> Michael Nielsen [17]

The “scientific knowledge” mentioned in the above quote encompasses any kind of knowledge including problem definitions, ideas, code, data, methods, and journal articles. The goal is to allow for more opportunities for collaboration and sharing of information by having this information available early enough so that it is useful to others. This can prevent expensive duplication of effort and encourage the direct comparison of results that is necessary for meta-analysis studies.

For computer-science research, since the research products are often not just papers nor data, but software implementations of algorithms, direct comparisons in terms of metrics such as accuracy and speed must be performed on a variety of datasets and machines. Computer science is not the only field that makes heavy use of software development: according to a survey of UK researchers by the Software Sustainability Institute which spanned disciplines as diverse as social science, medicine, and engineering, as high as 56% of researchers implement their own research software, but 21% of this subset had no training in software development [6].

Since software development is already a large part of the research process, it can be argued that a review process of scientific software is just as important as the paper peer review process. There are already issues with papers not having enough statistical power to present reproducible results, but these problems can be understood through careful reading of the experimental methods [18, 19]. Problems in the software are harder to analyze based on the papers alone and these have been known to result in paper retractions [20–22].

This lack of training in software engineering has prompted many open science advocates to offer open access teaching materials so that training is available to everyone. In particular, the Software Carpentry organization has published several guidelines and workshops to teach best practices for handling software and data that are relevant to scientific research [23, 24]. These include using traditional software engineering tools and practices such as version control systems, build automation, issue trackers, and pre-merge code reviews.

1.1.3 Open neuroscience

The history of biology has been defined by the tools used to visualize various biological structures and processes across many scales [25]. Neuroscience, in particular, has made use of many techniques: from the patch clamp for studying ion channels to EEG for recording brain surface electrical activity to fMRI and fluorescent microscopy for observing brain and neuron activity respectively, these techniques have collectively allowed neuroscientists to characterize what they are looking at. These techniques have made quantitative image analysis a large part of biology when measuring and comparing results across different samples. However, the results taken from different laboratories may not be comparable due to variability in laboratory protocols [26] or simply due to insufficient access to raw data necessary to accurately use Bayesian inference for predictions [27].

To address this problem, projects such as the Neuroscience Information Framework (NIF) [28] and Neuroimaging Informatics Tools and Resources Clearinghouse (NITRC) [29] provide databases of neuroscience data and tools that are machine-readable. These are part of recent research efforts to improve brain mapping and modelling [30, 31]. Some of these efforts are based on manual data annotation [32–35], while others such as the BigNeuron project from the Allen Institute for Brain Science are focused on completely automated reconstruction. This last project is the subject of Section 1.1.4.

1.1.4 BigNeuron

The secondary scientific outcome is to provide an implementation of ORION3 for the BigNeuron project. This requires using the common interface provided by the Vaa3D biomedical imaging toolkit [36–38]. This toolkit allows biologists to visualize and analyze biomedical imaging datasets. Vaa3D can be extended through the development of plugins. This allows algorithm developers to make their interactive and non-interactive methods for image analysis available for biologists to use without having to switch between multiple programs for processing and visualizing data.

Creating an image analysis tool that can be easily integrated into other systems allows others to reproduce the results in order to compare with both ground truth data and other algorithms. There has been an attempt at comparing neuron-morphology extraction algorithms in the past under the DIADEM Challenge which contributed datasets and a metric for comparing the neuron tracings from algorithms against tracings from gold standard reconstructions [39–41]. The DIADEM Challenge used six datasets from different neuroscience institutions in order to have a diversity in terms of source of the neurons (i.e., from different species and structures from different brain regions) and in terms of the laboratory protocols (i.e., different labelling and microscopy techniques).

The DIADEM Challenge was successful in raising awareness of the problem of neuron reconstruction and several new 3D reconstruction methods and metrics have been proposed after the end of the DIADEM Challenge (see Section 1.5). However, the lack of larger public datasets, standardized metrics, and readily available algorithms have made it difficult to compare new methods for neuron reconstruction.

To solve this problem, an open-science project called BigNeuron [42, 43] continues where the DIADEM Challenge left off. Instead of comparing the output on a few datasets, as in the DIADEM Challenge, BigNeuron aims to enable high-throughput analysis of neuron microscopy stacks using multiple methods contributed from various algorithm developers. By standardizing on the Vaa3D platform, this precludes any issues that might arise due to differences between how each method handles data which means that all the algorithms can be bench-tested at once without a need for translating data between formats.

However, since the Vaa3D is written in C++, incorporating plugin code written in non-native languages poses a problem. For this reason, the BigNeuron project organizers recommend that all algorithms that are submitted be in C or C++. From the BigNeuron FAQ [44]

> Q3. **How can I incorporate code written in Matlab, Java, Python or another language other than C/C++?**
>
> A. BigNeuron is a very large scale project, and enforcing a unified API is critical to ensure fair comparison for any pre-defined assessment. We thus discourage usage of Matlab, Java, Python or other programming languages besides C/C++ for this bench testing.

This is why a rewrite of the code is necessary rather than trying to integrate MATLAB via the MATLAB Engine Interface. Further discussion of the benefits of this decision are discussed in Section 1.4.

1.2 Systems Development Life Cycle

Before starting with the development, we need to outline the general steps needed to achieve the above stated outcomes and deliverables. In systems engineering, these steps are called the Systems Development Life Cycle (SDLC) of the project [45]. There are many variations of the SDLC each suited to different kinds of projects; Figure 1.1 depicts a simple version used in this project with six phases:

Planning: In this phase, project management and resource allocation details are determined. This includes scheduling, tools, and defining project objectives;

Analysis: This phase analyzes the project requirements and defines specific technical milestones that relate to the objectives defined in the Planning phase;

Design: The parts of the system are delineated and the expected input/output characteristics of each part are determined;

Implementation: The physical system is built during this phase using the system design from the previous phase;

Testing: As the Implementation phase proceeds, each standalone part is tested individually in what are known as unit tests. When the parts interact with each other, integration tests are conducted to determine that these parts are compatible;

Maintenance: In this phase, the system is put in production and monitored for changes in system performance and project requirements. When these changes necessitate an update to the system, the project may return to the Planning phase.

It is important to note that the phases of this life cycle are not discrete; there is overlap between phases. For example, parts of the design may change as the implementation continues as more information about the physical system is available.

1.3 System objectives

As opposed to the scientific outcomes discussed in Section 1.1, the system objectives are the engineering deliverables; these are more closely tied to the Design and Implementation phases as these objectives influence decisions made

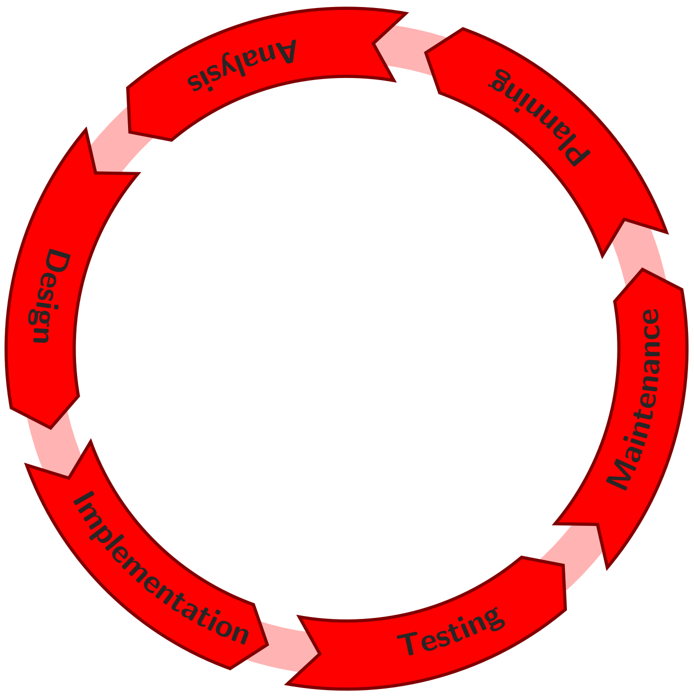


Figure 1.1: **A simple systems development life cycle (SDLC)**: This figure depicts an example of a life cycle used to delineate the phases of development.

about the underlying resources and system architecture.

The *first objective* is a complete conversion of the existing code from MATLAB to native code which will be referred to as ORION^{m} and ORION^{c} respectively. This involves an analysis of the existing code to see which parts of the algorithm will be converted.

The *second objective* is the ability to easily integrate the system with Vaa3D as a plugin. This requires looking at the interface that Vaa3D uses and creating compatible data structures so that the data does not need to converted between multiple formats in memory.

The *third objective* is a test suite to verify that the components of the ORION^{c} system operate on the same input and produce expected outputs which are comparable to the ORION^{m} code and follow expected properties outlined in the design.

The *fourth objective* is to ensure that the system provides a means for reproducibility by testing the software under different conditions and making it possible to the replicate the software environment so that others may run the software.

1.4 Benefits

The first objective requires that all of the MATLAB code be replaced by native code. This provides several benefits:

a) it removes a dependency on MATLAB which requires that all users either install a licensed copy of MATLAB or use the MATLAB Compiler Runtime for deployment;

b) it provides the benefit that changes in function behavior between versions of MATLAB do not effect the output of the code; and

c) it implements certain operations that can run faster in native code than in MATLAB.

The second objective will allow the code to work with Vaa3D, a widely used tool for visualization and analysis. Furthermore, integrating with one tool will provide a framework for integrating with other biomedical image analysis tools such as ImageJ [46].

The third objective provides a safety net so that current and future development clearly defines the expectation of the code not only in the documentation, but as executable tests that indicate when a change causes these expectations to no longer be met.

The fourth objective is what ensues that the previously discussed benefits are available for others to use on their machines and allowing for reproducibility of the neuron reconstruction results. This objective is what makes the project an “open-science” project.

1.5 Neuron reconstruction and tracing

Computational neuromorphology was first started in the late 1960s with initial attempts to capture microscopy images for computer storage and analysis [47, 48]. However, due to limitations in computer processing and storage, it was not until the late 1990s that a public dataset of neuron morphology was available [49]. With the release of the DIADEM Challenge data, there was an opportunity for more researchers to participate in the development of automated neuron reconstruction algorithms [40].

After the DIADEM Challenge concluded, there have been several papers that provide new approaches for neuron reconstruction. The papers can be grouped roughly into either segmentation-based reconstruction or seed point reconstruction methods. Segmentation-based reconstruction uses image features to determine tubular structures for labelling the volume and as such obtains the centerline and structure boundary at once. Seed point reconstruction works by obtaining seed points that are representative of the centerline and then connecting these seed points to reconstruct the neuron.

1.5.1 Segmentation-based reconstruction

In Bauer, Pock, Sorantin *et al.* [50], the authors present a method for automatic segmentation of 3D volumes of blood vessels especially where the vessel tree contains many overlapping structures. This is achieved by first extracting the tubular shapes using a Hessian filter. The results of the Hessian filter are then further processed by using a medialness filter to suppress false data (such as tumors or overlapping structures) that might occur at the boundaries of the vessel. Once the tubular structures are detected, the centreline is extracted by traversing the medialness response along the path with maximal value. The centreline and boundary provides structural information that can be used to construct model of the vessel structure that takes into account properties of blood flow (vessel radius, branching angle). This structural information is then used as a shape prior for assigning each tube to a tree in order to handle multiple trees.

In Jiménez, Papadakis, Labate *et al.* [51], the authors present a method that first uses low-pass, high-pass, and Laplacian filter responses as features to train an SVM to classify voxels as being background or foreground. This is used to binarize the neuron data so that only a subset of the volume needs to be analysed for seed selection. Seed selection is performed by applying a distance transform to the binary volume and then only keeping voxels that have a greater distances than the average distance in the 26-neighborhood of that voxel. Tracing between these seed points is then done with a variation of Dijkstra's algorithm that adds weights based on the distance transform and on the orientation between each successive edge. These constraints keep the centerline path from deviating from the center of the tubular structure and from having sharp changes in direction.

Hernandez-Herrera, Papadakis and Kakadiaris [52] describes a segmentation method based on one-class classification. This segmentation method uses the Laplacian filter response in order to create a training set of background voxels. Two representative eigenvalues of the Hessian filter are then used to create a discriminant function based on the distribution of these eigenvalues. This discriminant function is then used for segmentation.

1.5.2 Seed point reconstruction

In Xie, Zhao, Lee *et al.* [53, 54], the authors cover a method for tracing based on choosing seed points that represent the underlying neuronal structure and then connecting these seeds to produce a neuron tree. The seeding is initiated by applying a windowing cube on the volume in order to detect seed candidates at the local maxima. These seeds candidates are pruned to simplify tracing. Then a cost function is introduced in order to a find a path between the seed points. This cost function incorporates a smoothness term which constrains new edges between seed points so that large changes in orientation are avoided. Finally, these local edges are used to construct a global tree by using the minimal-spanning-tree (MST) algorithm.

Luo, Sui, Wang *et al.* [55] presents a method for neuron reconstruction that is based on an open-curve snake model. This method starts with a seed detection step that is based on what is known as a Sliding Volume Filter which selects a spherical region around each voxel in order to calculate the orientation of the gradient vectors within that region relative to center of the spherical region. By calculating the filter response as the cosine of the orientation, regions with a high response have more voxels where the gradient is oriented towards the center which indicates a tubular region. This filter response is used for initial seed point selection. These seed points are further pruned by using the eigenvalues of the Hessian of the volume to choose points that lie along the centerline of the tubular structure. The intensity of the voxels at the seed points is recalculated to create an SVF-enhanced image which is then used by an open-curve snake model for neuron tracing. This model creates a deformable model that is used to add edges to a curve that extends along the direction of the eigenvalue of the Hessian calculated earlier.

Gulyanon, Sharifai, Bleykhman *et al.* [56] use a joint probability model to both optimize voxel labels (foreground, background) and open-curve snake model configurations. This probability model uses the Frangi vessellness features as a data prior. A Discriminative Random Field is used to determine initial labels for the voxel classifications and voxels with a high confidence are used as the initial seed points for the 3D snake configuration. This inference is performed on subvolumes so that the probability model only uses the local intensity distribution.

1.5.3 Neuron tree similarity metrics

The DIADEM Challenge also provided a metric for comparing tracings from an automated reconstruction to gold standard tracings from human experts [41, 57]. This method works on the SWC format which provides both a spatial location and a radius for each point in the neuron tree. The method of registering the test tracing to the gold standard tracing starts by choosing an unregistered gold standard bifurcation node. To check that there is a matching node in the test tracing, the method looks for a test node that is in within a cylindrical region around the same spatial location as the gold standard node in order to find a match. The matching continues along both the parent and any children of the gold standard node. In order to determine that a gold standard node and test node are sufficiently close, a path length error is calculated by taking the path length between a nodes in the test tracing divided by the corresponding path length in the gold standard tracing. As long as this path length error is within acceptable bounds, the path is considered a match. Finally, if there are any gold standard nodes that have not been registered, the algorithm tries to determine if there is any already registered path that passes through that node in order to determine a match.

Recently, a new method for measuring the similarity of neuron tracings has been proposed in [58, 59]. This method measures the similarity between two tracings by using a Gaussian-weighted distance field between a given point in one tree and the closest point in the other tree. This allows for small deviations between the two trees without excessive penalization. This similarity metric is then applied to every point in one tree in order to determine a graph coloring that assigns each node in one tree to a corresponding node in the other tree.

There are still many more challenges for working with neuromorphology data including working with time-lapse data, handling subcellular structures, working with dense trees that contain multiple neurons, and dealing with electron microscopy data [39]. The target of all these challenges is to capture the dynamics of large neural circuits in hopes that understanding how groups of neurons work will provide insight into brain function.

“Measure twice and cut once.”

Carpentry rule of thumb

“But cut ting is more fun than m eas uri ng!”

Anonymous

2

Planning and Analysis

Before writing a single line of code, it is a good practice to understand the scope of the problem. Gathering the deliverables and requirements of the project is necessary not only for understanding how long the project will take, but also the order in which to implement each component. Establishing this order proves very useful for later in the testing phase of the project (Chapter 5).

2.1 Design principles

In order to meet the objectives listed in Section 1.3, certain principles need to be agreed upon before the design phase can begin. These principles are meant to direct how resources are used in the design and implementation.

The first of these is to keep the project history in version control and make the project publicly available as soon as possible. The choice made here is to use the Git version control system with the GitHub hosting service (`https://github.com/`) since it has been adopted by many other similarly scoped science projects including Vaa3D, OpenCV, and InsightToolkit; this gives it easy visibility via search engines. This is to address the open-science portion of the scientific outcome as it allows for a workflow that allows work-in-progress changes to reside alongside the stable codebase so that even work that is not yet complete is available as soon as possible.

The second principle is to choose an implementation language that makes integration easy (i.e., the second engineering objective). The choices for native languages that are widely available for this purpose are C and C++. Due to the way C++ implements naming conventions through *name mangling*, this can complicate the integration process and make the library maintenance more challenging due to changes in the ABI. To mitigate this, one common approach to implement a C wrapper API/ABI around an existing C++ API in what is known as an hour-glass interface [60]. This provides a stabilized ABI which means that if libraries are updated independent of the ORION^{c} code, then the ORION^{c} code will not have to be recompiled. However this can complicate development by having to maintain two layers, so to keep the development simplified, the main implementation language is in C.

As with all software, the ORION^c software has a series of steps required to prepare the software for building and installing and since ORION^c is native code, these steps can be complex because it needs to run on multiple platforms. Therefore, it is expedient to create an automated build system that outlines the steps needed to build ORION^c so that the process of creating the final binaries is replicable by anyone that obtains ORION^c and reduces the need for reading manual instructions. This automated build system should also be able to run tests to ensure that the software works as expected. The ORION^c project uses GNU Make since it is a portable build tool that runs on many systems. Overall, this specific principle improves reproducibility.

Another principle that is used to guide the development is to avoid premature optimization; that is, do not attempt to make the code more efficient before it is necessary. Instead of guessing which parts of the code are slow, a profiler can be used to measure the bottlenecks in the code. It is also not advisable to focus on optimization at the stage of rewriting the software.

This principle does not give the developer a license to write code in a way that is not optimizable later. For example, at this stage of the project, parallel computing will not be implemented through the use of tools such as POSIX threads and OpenMP since it can complicate debugging and testing. However, through careful avoidance of programming methods that make parallel programming difficult, such as global variables and file system access, later refactoring of the code to use a parallel architecture is possible.

This rewrite should be approached in an incremental manner by using test-driven development. For example, instead of trying to convert many modules of ORION^{m} at once, each is converted one at a time after writing tests for that specific component to ensure that it works independently. This also means that adding new external dependencies is done at the last possible moment. If a module of ORION^{c} requires an FFT function, the function should be created first as an empty stub function that is used to understand the minimum number of parameters needed to implement the FFT functionality. Then when the external dependency is incorporated, the call to the external library can be placed in the stub. This helps avoid creating coupling with the external library so that different approaches can easily be tried by only having to change the code at a single spot.

2.2 Challenges and risks

While the code for the algorithm already exists, starting with a line-by-line translation of the MATLAB code has some limitations outlined as follows.

Toolbox: The code is written to use MATLAB's extensive specialized toolboxes for image processing and statistics which means that equivalents must be incorporated into the new codebase.

Memory management: Since MATLAB is a dynamic array language with automatic memory management, it is simple to create multidimensional array and extend it without having to keep track of the variable's size or the variable lifetime. Since C uses manual memory management, it is necessary to manually allocate and release memory to avoid memory leaks.

Data layout differences: MATLAB uses column-major and 1-based indexing, while C/C++ both used row-major and 0-based indexing. Some of the code will be written with the assumption that all indices start at 1 and this may not be documented everywhere. This is discussed in more detail in Section 4.1.

Caching: The ORION$^{\mathrm{m}}$ code makes frequent use of the file system to cache calculations between runs. The purpose of this is to speed up experiments so that when an experiment is rerun, any images that have been processed in an earlier stage (i.e., segmentation) do not need to be reprocessed in later stages (i.e., centerline extraction). Code written in this form imposes an algorithm structure that is no longer strictly imperative — the code is now interspersed with checks to see if the data already exists and instead of passing the data between functions using multidimensional arrays as parameters, the parameters to the functions are filenames.

Subvolume: The MATLAB code breaks up the input data into subvolumes. This allows the computation to run a small region of the data which allows for processing data that may be too large to fit entirely memory. Furthermore, when used in conjunction with the aforementioned caching, the steps used for each processing stage can be more granular which means that if any processing is incomplete (e.g., because the computer runs out of memory or disk space), the data is not entirely lost. However, this complicates the algorithm because any calculation involving coordinates in a volume must map indices in subvolumes to indices in the corresponding supervolume.

The Caching and Subvolume issues can both be described as design decisions that result in cross-cutting concerns. Cross-cutting concerns are parts of the program design that do not reside within only a subset of the system and cause dependencies between subsystems. These often manifest when an action must be taken in every module of a system. A classic example of this is logging — logging must be done in each module, but this requires that logging metadata must be persisted so that each module can use it. This persistence adds an input to each module that does not strictly relate to the function of that module. In the same way, caching and handling of subvolumes both require that each step read all subvolumes of the previous step and write all the subvolumes that will be used in the next step. This filename-based coupling makes it difficult to treat each step as a self-contained module. By removing these concerns altogether in the new design, the ORION^{c} system will be more extensible and modular.

2.3 Roads not taken

There were two possible ways to avoid having to do a rewrite that could have fulfilled some of the objectives listed above. These were not chosen because they would have not met the primary scientific outcome (open-science) nor the fourth objective (reproducibility). Both of these approaches allow for calling MATLAB from C code and thus allow for processing with ORION$^{\mathrm{m}}$.

The first of these approaches is to use the standard *MATLAB Engine API* that comes with MATLAB. This allows for controlling a MATLAB instance using inter-process communication which means each run requires starting a MATLAB process with a valid MATLAB license. This means that ORION$^{\mathrm{m}}$ can not be run multiple times as each run will use an additional MATLAB license. This precludes using ORION$^{\mathrm{m}}$ on a cluster as each additional process will fail when the number of licenses has run out. Furthermore, the startup time for each MATLAB process is significant enough to slow down each volume processed. This does not meet the reproducibility criteria of the fourth objective as it requires the end-user to have a license that may be difficult to obtain and furthermore can tie the software to a specific version of the MATLAB software.

The second approach is to use the MATLAB Compiler tool to generate a dynamic library (e.g., `liborion3mat.so` on GNU/Linux and `liborion3mat.dll` on Windows) from the ORION$^{\mathrm{m}}$ code that can be linked to C/C++ code. This allows for distributing MATLAB code to people that do not have MATLAB by using the MATLAB Compiler Runtime along with an encrypted archive of the ORION^{m} workspace. Unlike the MATLAB Engine API, this can be used in parallel. However, the encrypted archive is tied to a single operating system (e.g., Windows, GNU/Linux) and computer architecture (e.g., i386 or x86-64). This makes reproducibility difficult since the dynamic library can only be generated on the same kind of computer and version of MATLAB as the corresponding version of the MATLAB Compiler Runtime.

A preliminary prototype of the MATLAB Compiler approach for interfacing with MATLAB was attempted in the Analysis phase, but rejected as it made distribution difficult and added a large binary dependency that would need to be maintained in addition to the ORION^{m} code. This approach could work for a research project where the ability for others to run the code on different data sets is sufficient. This can be accomplished by creating a lightweight virtual machine that comes pre-installed with the library and MATLAB Compiler Runtime. Those without access to MATLAB will still be able to run the algorithm as is, but will not able to adapt it by modifying the code. However, we reject this approach in the context of ORION3 as it would violate the primary scientific outcome (open-science) since encrypted code prevents others from seeing how the algorithms for ORION^{m} and MATLAB are implemented.

Although we reject the above approaches, there is a benefit to using MATLAB. Writing native code rather than MATLAB code can make maintenance and reproducibility more difficult if the development does not aim for those goals early on [61, 62]. Many considerations go into creating portable software and many of these same considerations are also necessary for reproducible computational science (e.g., file system handling, floating-point accuracy, compiler differences, and dynamic libraries) [63]. The developers of MATLAB have already done much of the work required to make MATLAB portable across different operating systems. Fortunately, the ORION$^{\text{m}}$ code only needs simple file system handling support. Other issues such as the aforementioned floating point accuracy and compiler differences are still unresolved by MATLAB itself and must be addressed in ORION$^{\text{c}}$. Further discussion about these and other issues is Chapters 4 and 5.

2.4 ORION3 MATLAB call graph

Even if the ORION$^{\text{c}}$ code is a rewrite, it is a good idea to look at how the original ORION$^{\text{m}}$ codebase is arranged. One way to do this is to build a call graph, that is, a graphical representation of which functions are called by other functions. Using this, it is possible to recursively trace the execution of the code. By taking this call graph and creating equivalent functions in the native code, a direct comparison can be made between the two codebases. Thus, each function can be converted one by one.

There is a tool built in to MATLAB to create call graphs called `depfun`, however this tool runs slowly when running on the entire codebase. There is an alternative called `fdep` [64]. The results of `fdep` are then passed to the GraphViz graph layout tool to visualize the results [65]. Note that the call graph generated by `fdep` is generated using static call graph analysis, so functions that might not be called during execution may be included (e.g., function calls that are in dead code branches). The call graph for the MATLAB code is depicted in Figs. 2.1 to 2.3. In each of the figures, the label in the outermost box is a function name in ORION^{m} and each box inside is functions that are called from inside the body of that function. Arrows represent a function call. For example, in Fig. 2.1, the `readNegativeSamples` function calls `multiscaleLaplacianFilter` which in turn calls `Makefilter`.

Alternatively, the call graph represents the flow of data — the lowest levels deal with the least amount of data and as you go up higher in the graph more pieces of information are integrated together from many sources. An illustration of this can be seen in the relationship between data flow of `multiscaleLaplacianFilter` and `Makefilter` as sketched in Algorithm 1. What this shows is that a function lower in the call graph, `Makefilter`, is a simpler function in terms of information processing than its parent in the call graph, `multiscaleLaplacianFilter`, which has to integrate information from multiple calls to the `Makefilter` child function. This kind of relationship is generally the case between functions in a call graph and this is why it is often easier to test functions lower in the call graph first — there is less data to take into account and the functions lower in the call graph are generally more standalone. This means that they can easily be individually tested. This view allows for a testing strategy that starts from the bottom-most level of the call graph and steadily moves upwards.

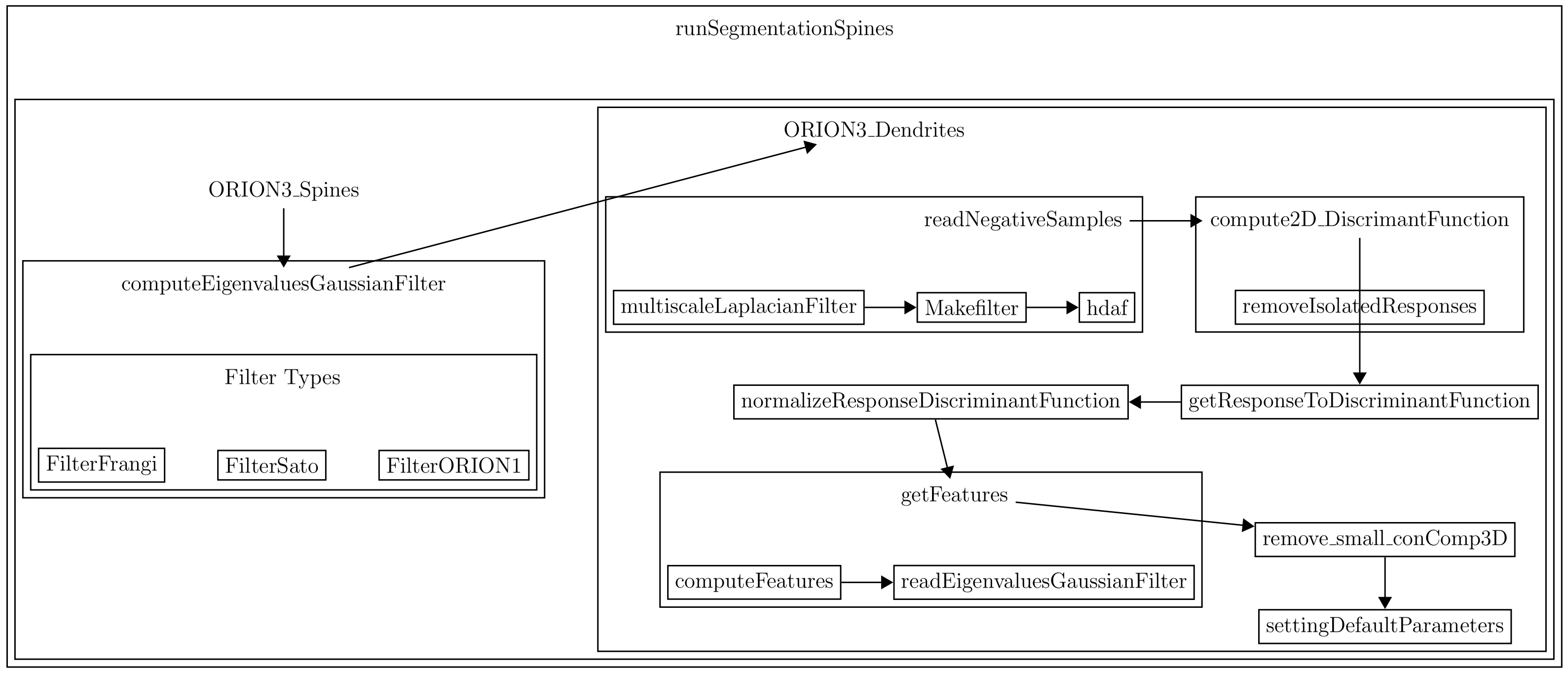
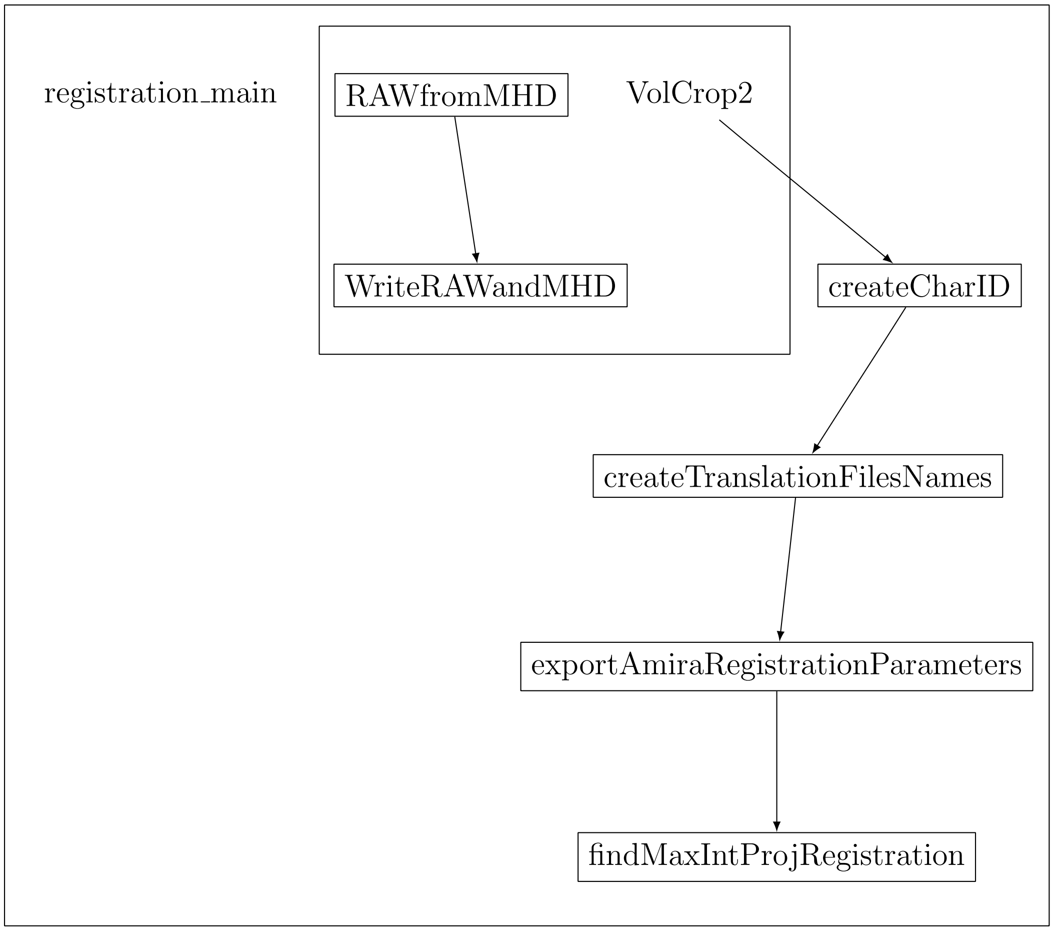


Figure 2.1: **Call graph of segmentation process in ORION$^{\mathbf{m}}$**

Algorithm 1 Sketch of `multiscaleLaplacianFilter` and `Makefilter` functions

```
1: function MAKEFILTER($filter\_size, degree, scale\_factor$)
2:     return $F$                ▷ Calculates a filter $F$ given the above parameters
3: end function
4: function MULTISCALELAPLACIANFILTER($Volume, degree, scales$)
5:     for $scale_i \in scales$ do
6:         $F_i \leftarrow$ MAKEFILTER($|Volume|, degree, scale_i$)
7:     end for
8:     $MaxScale(x, y, z) \leftarrow \arg\max_i (Volume * F_i)(x, y, z)$
9: end function
```



registration_main
RAWfromMHD
VolCrop2
WriteRAWandMHD
createCharID
createTranslationFilesNames
exportAmiraRegistrationParameters
findMaxIntProjRegistration

Figure 2.2: **Call graph of registration process in ORION$^{\mathrm{m}}$**

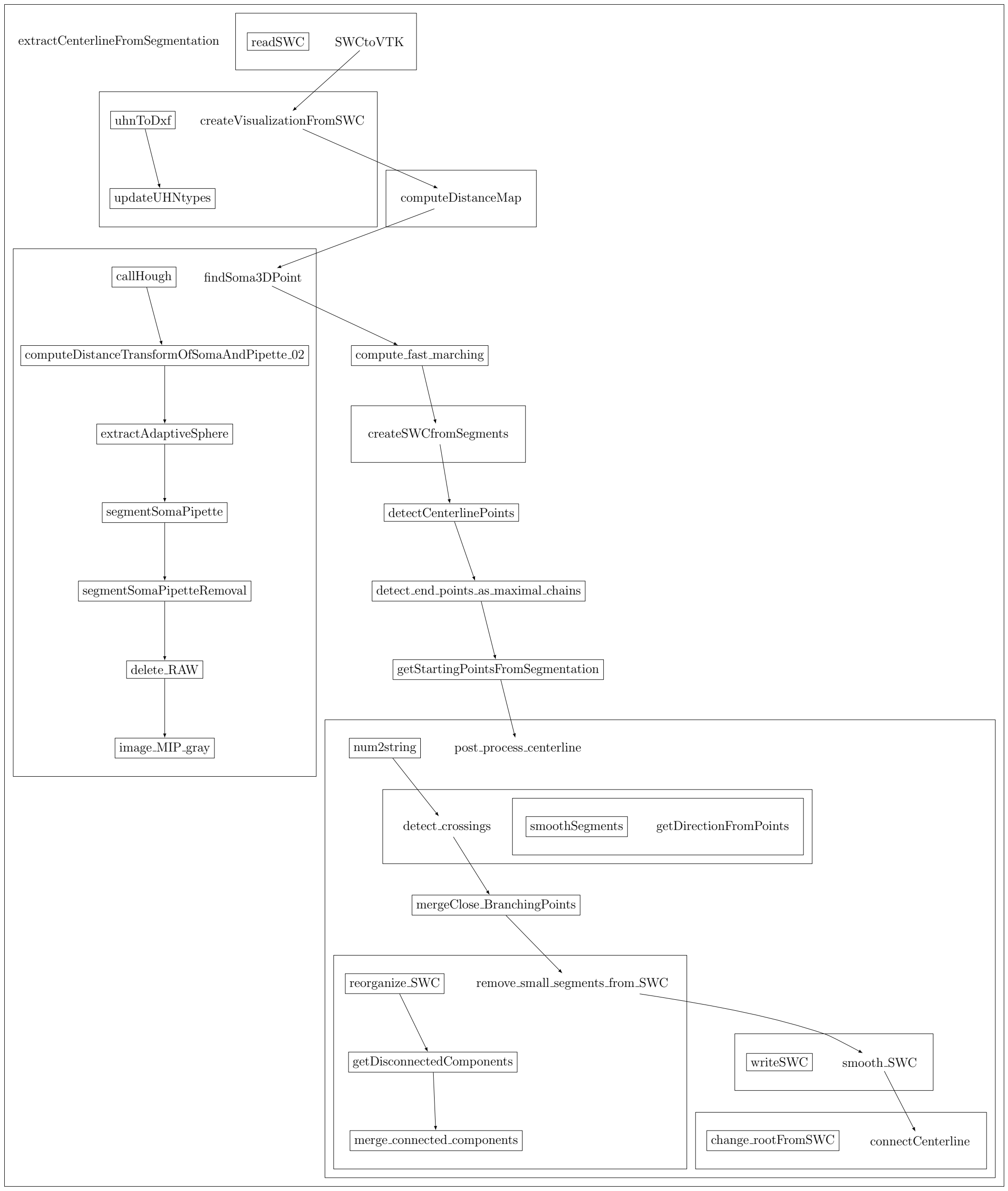


Figure 2.3: **Call graph of tracing process in ORION$^{\mathbf{m}}$**

“The trouble with computers is you *play* with them. They are so wonderful. You have these switches — if it’s an even number you do this, if it’s an odd number you do that — and pretty soon you can do more and more elaborate things if you are clever enough, on one machine.”

Richard Feynman in *Surely You’re Joking, Mr. Feynman!: Adventures of a Curious Character*, 1985

3 Design

Once the planning is done, the actual technical details of the project are determined in the Design phase. This phase is not a discrete step that is separate from the following Implementation phase; as the implementation continues, the Design is updated to take into account new information. As such, it is important that the Design is able to incorporate incremental changes otherwise adding new changes will become difficult — especially when fundamental data structures need to be modified. The following details these design decisions.

3.1 Incorporation of ORION^{m}

Since the algorithm already exists as a design in the ORION^{m} implementation, this can be leveraged as a starting point for the conversion. This means that the structure of ORION^{c} will start off with the same structure as the MATLAB version. This is to reduce the cognitive load when rewriting and testing each component because the inputs and outputs remain the same and keeping the names the same makes it easier to navigate between corresponding functions.

To facilitate this way of working, the filenames and directory structure of the original MATLAB code are copied verbatim: instead of using the `.m` extension for MATLAB files, the `.c` extension is used. Inside each of these `.c` files, a function signature is defined that matches the one found in MATLAB with the exception that the name is prefixed with `orion_` such that a ORION^{m} function named `hdaf` would appear as ORION^{c} function named `orion_hdaf`. This prefixing is common in C libraries as a way to provide an application-specific namespace for symbols. This helps avoid naming collisions where multiple libraries may define the same symbol and these multiple definitions will need to be disambiguated.

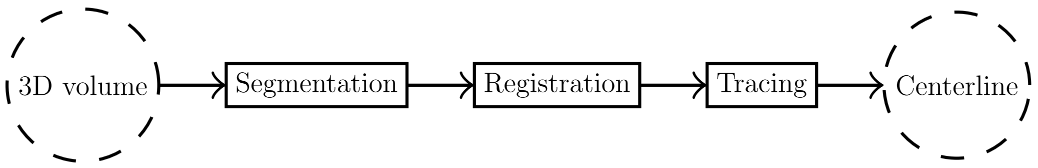


Figure 3.1: **High level diagram of ORION algorithm**: This high-level diagram shows that there are three steps to the ORION$^{\mathrm{m}}$ algorithm as described in the text. The input to the algorithm is a 3D volume of microscopy data and the output is a graph-based representation of the neuron morphology based on the centerline.

3.2 Algorithms and architecture

The architecture of a software system includes both the individual components and how they interact with one another and the core data structures that are used to transfer data between the components. The following description of the architecture will approach these details from the top-down.

The ORION algorithm as implemented in ORION$^{\mathrm{m}}$ consists of three parts (as shown in Fig. 3.1 where data is drawn as dashed-line ovals and processing is drawn as solid-lined rectangles), namely

Segmentation: to label the foreground and the background of the image (see Fig. 3.2);

Registration: for aligning subvolumes so that they can be used to create a single volume (this step is not needed in ORION$^{\mathrm{c}}$); and

Tracing: to extract a centerline from the volume to capture the underlying neuron morphology (see Fig. 3.3).

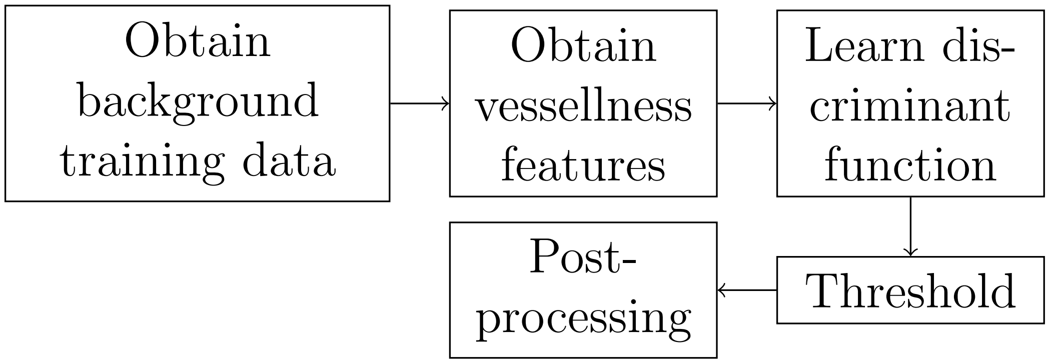

Obtain background training data
Obtain vessellness features
Learn discriminant function
Threshold
Post-processing

Figure 3.2: **Diagram of segmentation process**

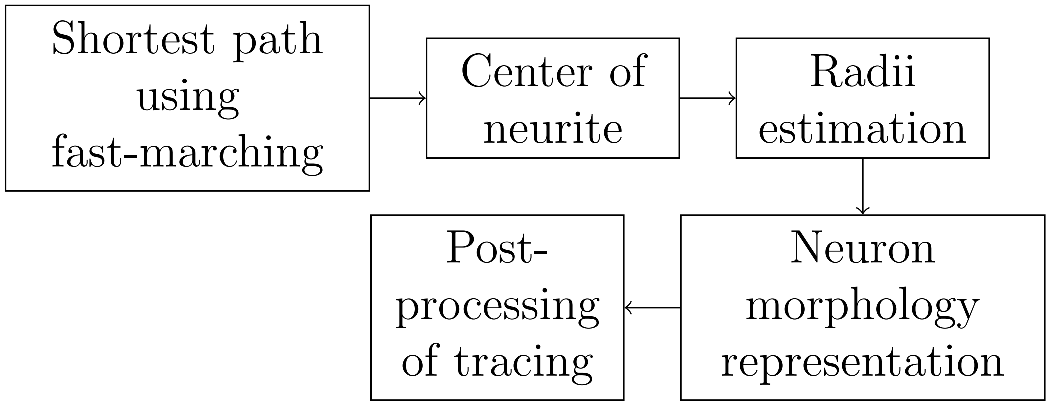

Shortest path using fast-marching
Center of neurite
Radii estimation
Neuron morphology representation
Post-processing of tracing

Figure 3.3: **Diagram of tracing process**

3.3 Anticipating change

Since the code for ORION^c changes rapidly throughout the project, it is necessary that the code structure is planned so that it does not have to change too drastically as new components are added to the project. This requires developing a project structure that does not require too many manual changes which slow down development. In the following, various aspects of the code structure and how they improve the code's ability to easily incorporate change.

3.3.1 Directory structure

The first part of setting up a new project is choosing a directory structure. This determines where new files should go. A common structure is to create separate directories for code for a library and code that will be compiled into an executable. This is reflected in the directory structure where

`lib`: contains subdirectories which have all the source code that will be compiled together to create a library file (i.e., `liborion.a` on many Unix systems);

`lib/t`: contains source files that have a matching directory layout to the rest of the `lib` directory so that it is easy to find the corresponding test to a given component (e.g., a library source file `lib/path/func.c` will have a corresponding test source file in `lib/t/path/func.c`); and

`src`: contains source files that will be compiled into executables that can be run at the command-line.

3.3.2 Build system

Whenever a native build system is chosen, an important property is that it should be portable. In order to fulfill this property, the build system is written using GNU `make` which is a portable version of the Unix `make` build tool. This tool works by reading a `Makefile` that lists the prerequisites for a given target file and a set of commands needed to build those prerequisites. If the target is older than any one of its prerequisites, then that target file is rebuilt. This allows for testing changes to large projects without out needing to rebuild the entire project.

Another property is that the build system should not have to require many changes when adding new source code to the project. This is accomplished by using automatic dependency scanning. Before any code is compiled, the files are scanned to create a list of prerequisites for each file. By scanning for prerequisites, most of the files in the project do not need explicit rules in the `Makefile` which means that when a new file is added to the project, no changes to the build system are necessary.

An automatic build system allows for easily building the project on a new machine. If the build system is portable and well-tested, the project should build on the new system without manual intervention. When making a project for scientific purposes, this is essential for reproducibility because the knowledge of how to build the software is completely written down in an executable form.

3.3.2.1 Configuration

Software projects do not live in isolation; many projects depend on outside libraries to implement functionality. However, whenever a dependency is added, this adds another point where the build could break. Outside libraries are not necessarily in the same location on every system. In order to portably build the ORION^c with these outside libraries, the build system has automatic build configuration that scans for the location of any dependencies that it needs. This allows the same build system to be used on multiple systems without having to use any fixed paths that are specific to a single computer.

In addition, sometimes developers may want to enable system-specific features in order to speed up their code. When this happens, the code must be able to detect when such features are available and use a fallback if they are not available. The build system for ORION^c contains an automatic system configuration scanner that it uses for this purpose. This configuration scanner is currently only used to enable the GCC compiler's branch prediction macro (`__builtin_expect`) which is used to give hints to the compiler whether or not a given condition is likely or not [66].

3.3.2.2 Debugging

As the project progresses, the development is inevitably going to come across bugs. When programming in C or C++, the class of bugs that occur are different from the kinds that occur in MATLAB. These usually are related to invalid memory access (e.g., buffer overflows, stack overflows). To help ease debugging, the build system must also support tools that can help track these errors. These tools often require the addition of build options that add extra metadata to the compiled code. The build system thus supports **code coverage builds:** which are used to determine how much of the code is tested by the testing code; **debug builds:** to enable debugging symbols that can be used to stop and inspect the execution of a program and catch memory access violations; and **profiling builds:** which can be used to understand the areas of the code that are running slowly in order to intelligently decide whether to optimize it.

“ One of my most productive days was throwing away 1000 lines of code. ”

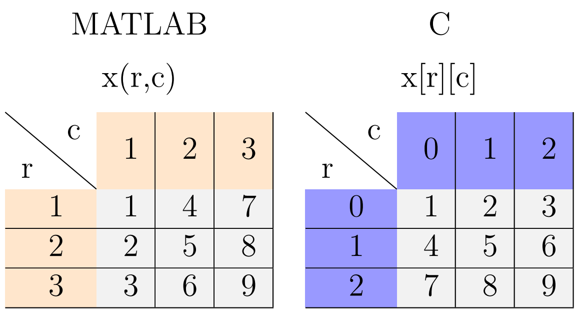
Ken Thompson

“ Programming, programming, all through the night,
We’re stuck here until our new program works right.
Programming, programming, isn’t it fun?
The maintenance starts when debugging is done! ”

Steve Savitzky in *The Programmer’s Alphabet*, 1981 [67]

4 Implementation

As discussed in Chapter 3, the Implementation stage starts by converting the MATLAB code to native code by following the same architecture as the ORION^{m} code as initial point. This chapter discusses the real world characteristics of the code used to implement ORION^{c}. This starts with the data structures that are used for calculations. Implementing a data structure that



MATLAB

x(r,c)

r \ c	1	2	3
1	1	4	7
2	2	5	8
3	3	6	9

C

x[r][c]

r \ c	0	1	2
0	1	2	3
1	4	5	6
2	7	8	9

Figure 4.1: Difference in memory layout between MATLAB (column-major) and C (row-major).

captures the domain of the problem correctly is the most important first step in any software system because once a data structure is chosen, it can be difficult to change as every part of the code relies on the properties of that data structure.

4.1 Data structures

The main data structure used in the implementation is an n-dimensional array or tensor. Since the algorithm is meant to work with 3D data, the n-dimensional array has a fixed $n = 3$. To define this data structure, we need to examine the memory layout. There are two approaches to this: column-major and row-major. The difference between the two is based on which dimension index changes the faster when accessing memory linearly. This difference is illustrated in Fig. 4.1 which depicts a data that is stored in memory with increasing value from 1 to 9. The indices to access the same value differ between MATLAB and C not only due to MATLAB's 1-based indexing and C's 0-based indexing, but also the memory layout.

For ORION^c, the n-dimensional array is stored as row-major as this is the default that most C programmers expect. This array is represented by the C structure

```
typedef struct {
 pixel_type* p;                    /* allocated memory for data */

 size_t sz[PIXEL_NDIMS];           /* size of each dimension    */
 pixel_type spacing[PIXEL_NDIMS];  /* physical spacing          */
} ndarray3;
```

where `pixel_type` is represents the floating point storage type used for the n-dimensional data, `p` is a pointer to the block of memory where the data is stored, and `PIXEL_NDIMS` is the constant 3. The `spacing` field is used to store the physical spacing between items in the n-dimensional grid and is used for normalizing calculations.

There are also other data structures that are mainly used for book-keeping and storage of the algorithm parameters. For example, to obtain the output of the Hessian filter, a collection of the eigenvalue features and the associated scales used for that filter are returned by using the C structure

```
typedef struct {
 /* scale used for this filter result    */
 float scale;
 /* the three eigenvalues for each voxel */
 array_ndarray3* eig_feat;
} orion_eig_feat_result;
```

This allows for access to the filter response as well as the original parameters that generated that response.

4.2 Numerical considerations

Scientific computing often uses floating point for calculations, but the techniques for reducing numerical error are often overlooked [68]. For example, a simple summation of a list of floating point numbers accumulates numerical errors as more numbers are numbers are added. To compensate for this numerical error, algorithms such as Kahan summation are used to accumulate this error in another *running compensation* variable [69].

Another numerical consideration is making sure that the result of the floating point operations can fit within the limits of the storage type. For example, multiplying large floating point numbers can cause the result to become larger than the largest `double` (64-bit) floating point value which means that the result is no longer representable. This can happen when calculating factorials for a Taylor series. To avoid this, it is important to calculate the bounds of a calculation and provide error checking for when the input could lead to invalid results. For a factorial function that works with 64-bit floating point data, we can calculate the bounds of input to the function $n!$ as $n \in [0, 170]$. In addition, since the domain of the input variable is so small, storing the pre-calculated data in a lookup table can help avoid numerical errors that accumulate through multiplication.

4.3 Prototyping components

As mentioned before, the design of ORION^{c} initially follows the architecture of the ORION^{m} code. To accomplish this, each MATLAB function has a corresponding C function that is located in a similarly named file. Using the call graphs obtained in Section 2.4, we first choose a function that is a leaf node in the call graph. This function requires very little data for its inputs, so testing the ORION^{c} implementation against the ORION^{m} implementation does not require a lot of set up for each of the parameters. In order to prototype the ORION^{c} function, we take a set of test parameters and apply them to the ORION^{m} function and capture its output. This output is then placed inside a test file and used to compare the C output to the MATLAB output. This procedure is then repeated for each parent function in the call graph.

4.4 Library integration

Some of the calculations needed for this implementation require outside libraries. In order to integrate with these libraries, it is necessary to convert the ORION^{c} data structures into appropriate data structures for each library.

To compute the vessellness filter, the ITK library is used [70]. This library is widely used in biomedical image analysis and has many implementations of filters that are commonly used for image segmentation. The ITK library uses a data structure called `itk::Image` for to store the inputs and outputs for calculations. This data structure is internally very similar to the ORION^{c} `ndarray3` data structure as they both use a row-major block of memory to store their data. As such, it is possible to convert the `ndarray3` data to `itk::Image` data without having to copy the data by sharing the buffer between the two libraries.

For computation of the fast-Fourier transform, a small library called Kiss FFT is used [71]. This library has functions for computing an n-dimensional Fast-Fourier transform (FFT). This library also uses a n-dimensional array data structure that is similar to `ndarray3` which makes it easy to write a wrapper to the FFT code that takes an `ndarray3` input.

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

Brian W. Kernighan and P. J. Plauger in

The Elements of Programming Style, 1978

5 Testing and verification

When testing a system, there are two related procedures that are used to ensure that the system is working in the intended manner: verification and validation. Verification is testing whether an implementation of a model is correctly implemented. This is analogous to asking “did we build it right?”. Validation is checking the accuracy of the model to a real system. This asks “was it the right thing to build?” [72].

When testing is done on the whole system to check if the final output makes sense, this is a kind of validation. This is how the ORION$^{\mathrm{m}}$ is currently tested. However, validation testing can not stand on its own. We also need to improve the system’s verification testing. One such way is to implement unit testing. This kind of testing involves running each unit of the system (such as a function) and checking if the expected outputs are generated for a given set of inputs.

By performing unit tests, the expectation of a given unit is recorded in a way that it can be run repeatedly and in completely automated manner. This allows for running the same tests both in new environments and whenever a part of the code changes. Since it is completely automated, debugging in case there are any unmet expectations can be performed quickly. However, testing can only prove the existence of bugs, not their absence. This is why a particular form of software engineering methodology called Test-Driven Development (TDD) advocates writing tests before the actual code being tested. In this way, the test code will fail first and just enough code is added to make the test pass. In this way, it is clear that the added code made that specific test case pass.

Some of examples of tests that are used in the ORION$^{\mathrm{c}}$ code include **analytic testing:** which test numerical calculations where an analytic solution is known for certain inputs (e.g., the FFT of $f(x) = \sin 2\pi x$); **property testing:** which tests if a function satisfies certain conditions on its outputs (i.e., a function that outputs data in sorted order can be checked to see if the sorted property is retained); **integration tests:** which test if two separate systems can work together (i.e., the output of one system can be used as the input to another system).

It is important to keep in mind that since many of these tests are performed on floating point data, the tests can not use strict equality and must compare their values to within a tolerance. For example, when testing the FFT implementation, one property that can be tested is if computing $\mathcal{F}^{-1}\left\{\mathcal{F}\left\{x\right\}\right\} = x$. However, due to numerical errors, the absolute error $|\mathcal{F}^{-1}\left\{\mathcal{F}\left\{x\right\}\right\} - x|$ is within a range of $\epsilon = 1 \times 10^{-7}$ when using 32-bit floating point numbers for computation — which is expected based on the machine epsilon [73, 74].

5.1 Testing portability

One of the issues with testing is that some tests that may hold one machine, may not hold on another. This can be due to differences in library versions, differences between compilers, or even differences between processors. This is why testing in multiple environments is necessary to ensure that the tests are themselves portable. In order to test in multiple environments, ORION[c] is tested using a continuous integration server which tests every change on a different machine using different compilers. Using a continuous integration server like this has helped track down some issues with using an older version of ITK and other issues that had to do with how a specific compiler interpreted the source code. Furthermore, using a continuous integration server allows for a workflow where every new feature can be worked on separately from the main “released” code and tested as it is being developed. Only when that code has been appropriately tested will that feature be brought into the main code. This allows in-progress code to be worked on separately from working code.

5.2 Tracing-based comparison with ORION$^{\mathrm{m}}$

In order to look more closely at whether or not the ORION$^{\mathrm{c}}$ code implements the same algorithms as ORION$^{\mathrm{m}}$, it is possible to compare the data from across the entire ORION$^{\mathrm{m}}$ pipeline. This can be accomplished by recording the input parameters given to every ORION$^{\mathrm{m}}$ function and passing those inputs to the ORION$^{\mathrm{c}}$ input. In Fig. 5.1, the pipeline for ORION$^{\mathrm{c}}$ is depicted on the top row and the pipeline for ORION$^{\mathrm{m}}$ is depicted on the bottom row. The middle row represents the process of taking the ORION$^{\mathrm{m}}$ data and passing it to the corresponding ORION$^{\mathrm{c}}$ component. Then the output of the components in the middle row and the bottom row are compared. The reason for this comparison procedure is so that the functions can be compared individually to find where the two systems deviate instead of having to see the differences accumulate over the entire system. A full description of the algorithm is given in Algorithms 2 and 3. In Algorithm 3, the algorithm makes reference to an “appropriate method”. This method depends on the kind of data being compared. For example, in Table 5.1, the comparison is done by generating a histogram a and b for each volume so that both histograms contain the same

Stack ID	histogram intersection
1	0.999973665203964
2	0.999967508148729
3	0.999978416844418
4	0.999982131154914
5	0.999959276433577

Table 5.1: Example of comparison captured from data for the `Makefilter` function

number of bins n and the same bin boundaries. Then the intersection of the two histograms is calculated as

$$K_{\cap}(a,b) = \sum_{i=1}^{n} min(a_i, b_i). \tag{5.1}$$

Algorithm 2 MATLAB tracing data capture

```
1: Set breakpoints at every  function start and end.
2: Run the MATLAB code under the debugger.
3: while  Program has not finished  do
4:     When the debugger stops, save the $State$ of the input and output data
       at each breakpoint along with a stack frame ID. Add $State$ to $SavedStates$.
5:     Continue the debugger.
6: end while
```

Algorithm 3 MATLAB tracing data comparison

```
1: $SavedStates \leftarrow$ the list of all saved states from Algorithm 2.
2: for $State \in SavedStates$ do
3:     Load the MATLAB data from $State$.
4:     Convert MATLAB data structures to C data structures.
5:     Call corresponding ORION$^c$ function using C data structures
6:     Compare the results using an appropriate method.
7: end for
```

$$\begin{tikzcd}
 & fun_1^C \arrow[r] & fun_2^C \arrow[r] & fun_3^C \arrow[r] & fun_4^C \arrow[r] & fun_5^C \\
\text{Input} \arrow[ur, bend left] \arrow[r] \arrow[dr, bend right] & fun_1^{M-C} & fun_2^{M-C} & fun_3^{M-C} & fun_4^{M-C} & fun_5^{M-C} \\
 & fun_1^M \arrow[r] \arrow[ur] & fun_2^M \arrow[r] \arrow[ur] & fun_3^M \arrow[r] \arrow[ur] & fun_4^M \arrow[r] \arrow[ur] & fun_5^M
\end{tikzcd}$$

Figure 5.1: Abstract diagram of pipeline of both ORION^{c} and ORION^{m}

“ The purpose of computing is insight, not numbers. ”

Richard W. Hamming in *Numerical Methods for Scientists and Engineers*, 1962

“ There’s no sense in being precise when you don’t even know what you’re talking about. ”

John von Neumann

6 Conclusion

The main contribution of this thesis is a neuron-morphology reconstruction system with an automated test suite. The existence of an automated test suite allows for future refactoring and additions to the code with confidence that each component behaves as expected. The consequences of this contribution go beyond just the testing of the current state of the system — testing allows the code to evolve to incorporate new designs.

While the code currently relies heavily on the original ORION$^{\text{m}}$ design, removing much of the MATLAB specific design decisions such as caching make it easier to refactor because the only input data to a ORION$^{\text{c}}$ function are those which are passed in directly as input parameters. This also leaves room for making parts of the system run in parallel as there will be no extra overhead or file system contention involved in reading and writing to the disk.

Furthermore, although the main backbone of the ORION$^{\text{c}}$ pipeline uses the same design as ORION$^{\text{m}}$, the implementation inside each component calls out to generic, reusable code. This reusable code (everything outside of the `lib/kitchen-sink` directory) is designed so that it can more easily be tested than the code based on ORION$^{\text{m}}$.

Future work on this software system can also include packaging the software for easy installation in repositories such as NeuroDebian [75]. This will allow end-users an easy way to install the software without having to deal with building it themselves. This also serves as another way of testing the software because these end-users will likely use the software on a wide variety of data as well as under different environments. This opens the door to improvements both to the robustness of the reconstruction algorithm and the portability of the software system.

Bibliography

[1] A. Santamaría-Pang, P. Hernandez-Herrera, M. Papadakis, P. Saggau and I. A. Kakadiaris, “Automatic morphological reconstruction of neurons from multiphoton and confocal microscopy images using 3D tubular models”, *Neuroinformatics*, vol. 13, no. 3, Jan. 2015. DOI: 10.1007/s12021-014-9253-2.

[2] D. Baltimore, in *The Invisible Future: The Seamless Integration of Technology into Everyday Life*, P. J. Denning, Ed., New York, NY, USA: McGraw-Hill, Inc., 2001, ch. How Biology Became an Information Science, pp. 43–55.

[3] MATLAB, *Version 8.1 (R2013a)*. Natick, Massachusetts: The MathWorks Inc., 2013.

[4] J. Spolsky. (6th Apr. 2000). Joel on software: Things you should never do, part I, [Online]. Available: http://www.joelonsoftware.com/articles/fog0000000069.html (visited on 4th Aug. 2015).

[5] S. M. Baxter, S. W. Day, J. S. Fetrow and S. J. Reisinger, “Scientific software development is not an oxymoron”, *PLoS Computational Biology*, vol. 2, no. 9, pp. 0975–0978, 2006. DOI: 10.1371/journal.pcbi.0020087.

[6] S. Hettrick, M. Antonioletti, L. Carr, N. Chue Hong, S. Crouch, D. De Roure, I. Emsley, C. Goble, A. Hay, D. Inupakutika, M. Jackson, A. Nenadic, T. Parkinson, M. I. Parsons, A. Pawlik, G. Peru, A. Proeme, J. Robinson and S. Sufi, *UK research software survey 2014*, Dec. 2014. DOI: 10.5281/zenodo.14809.

[7] G. K. Sandve, A. Nekrutenko, J. Taylor and E. Hovig, “Ten simple rules for reproducible computational research”, *PLoS Computational Biology*, vol. 9, no. 10, P. E. Bourne, Ed., e1003285, Oct. 2013. DOI: 10.1371/journal.pcbi.1003285.

[8] J. T. Leek. (Nov. 2013). How to share data with a statistician, [Online]. Available: https://github.com/jtleek/datasharing (visited on 28th Nov. 2015).

[9] A. E. Jaffe, T. Hyde, J. Kleinman, D. R. Weinbergern, J. G. Chenoweth, R. D. McKay, J. T. Leek and C. Colantuoni, "Practical impacts of genomic data "cleaning" on biological discovery using surrogate variable analysis", *BMC Bioinformatics*, vol. 16, no. 1, p. 372, 2015. DOI: 10.1186/s12859-015-0808-5.

[10] H. Wickham, "Tidy data", *Journal of Statistical Software*, vol. 59, no. 10, pp. 1–23, 2014. DOI: 10.18637/jss.v059.i10.

[11] R. Anderson. (2000). The end of DLL hell, [Online]. Available: http://web.archive.org/web/20010605023737/http://msdn.microsoft.com/library/techart/dlldanger1.htm.

[12] D. Burrows. (15th Jun. 2005). Modelling and resolving software dependencies, [Online]. Available: https://people.debian.org/~dburrows/model.pdf.

[13] P. J. Guo and D. Engler, "CDE: Using system call interposition to automatically create portable software packages", in *Proc. USENIX Annual Technical Conference*, Portland, OR, 2011, p. 21.

[14] National Digital Information Infrastructure and Preservation Program, "Preserving.exe: Toward a national strategy for software preservation", Library of Congress, Tech. Rep., 18th Oct. 2013, pp. 1–42.

[15] D. Thain, P. Ivie and H. Meng, "Techniques for preserving scientific software executions: Preserve the mess or encourage cleanliness?", in *Proc.* 12^{th} *International Conference on Digital Preservation*, Chapel Hill, USA, Nov. 2015. DOI: 10.7274/R0CZ353M.

[16] H. Meng, M. Wolf, P. Ivie, A. Woodard, M. Hildreth and D. Thain, "A case study in preserving a high energy physics application with Parrot", *Journal of Physics: Conference Series*, Dec. 2015, forthcoming.

[17] M. Nielsen. (28th Jul. 2011). Definitions of open science?, [Online]. Available: https://lists.okfn.org/pipermail/open-science/2011-July/000907.html (visited on 28th Nov. 2015).

[18] J. P. A. Ioannidis, "Why most published research findings are false", *PLoS Med*, vol. 2, no. 8, e124, 2005. DOI: 10.1371/journal.pmed.0020124.

[19] K. S. Button, J. P. A. Ioannidis, C. Mokrysz, B. A. Nosek, J. Flint, E. S. J. Robinson and M. R. Munafò, "Power failure: Why small sample size undermines the reliability of neuroscience", *Nature Reviews Neuroscience*, vol. 14, no. 5, pp. 365–376, Apr. 2013. DOI: 10.1038/nrn3475.

[20] G. Miller, "Scientific publishing. a scientist's nightmare: Software problem leads to five retractions", *Science*, vol. 314, no. 5807, pp. 1856–1857, Dec. 2006. DOI: 10.1126/science.314.5807.1856.

[21] Z. Merali, "Computational science: . . . error.", *Nature*, vol. 467, no. 7317, pp. 775–777, 2010. DOI: 10.1038/467775a.

[22] L. N. Joppa, G. McInerny, R. Harper, L. Salido, K. Takeda, K. O'Hara, D. Gavaghan and S. Emmott, "Troubling trends in scientific software use", *Science*, vol. 340, no. 6134, pp. 814–815, May 2013. DOI: 10.1126/science.1231535.

[23] G. Wilson, "Where's the real bottleneck in scientific computing?", *American Scientist*, vol. 94, no. 1, p. 5, 2006. DOI: 10.1511/2006.57.3473.

[24] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White and P. Wilson, "Best practices for scientific computing", *PLoS Biology*, vol. 12, no. 1, J. A. Eisen, Ed., e1001745, Jan. 2014. DOI: 10.1371/journal.pbio.1001745.

[25] P. Moore, *Visualizing the invisible: Imaging techniques for the structural biologist*. Oxford University Press, 2012.

[26] R. Parekh, R. Armañanzas and G. a. Ascoli, "The importance of metadata to assess information content in digital reconstructions of neuronal morphology", *Cell and Tissue Research*, vol. 360, no. 1, pp. 121–127, Feb. 2015. DOI: 10.1007/s00441-014-2103-6.

[27] R. A. Poldrack, "Inferring mental states from neuroimaging data: From reverse inference to large-scale decoding", *Neuron*, vol. 72, no. 5, pp. 692–697, 2011. DOI: 10.1016/j.neuron.2011.11.001.

[28] D. Gardner, H. Akil, G. A. Ascoli, D. M. Bowden, W. Bug, D. E. Donohue, D. H. Goldberg, B. Grafstein, J. S. Grethe, A. Gupta, M. Halavi, D. N. Kennedy, L. Marenco, M. E. Martone, P. L. Miller, H.-M. Müller, A. Robert, G. M. Shepherd, P. W. Sternberg, D. C. Van Essen and R. W. Williams, "The Neuroscience Information Framework: A data and knowledge environment for neuroscience", *Neuroinformatics*, vol. 6, no. 3, pp. 149–160, Sep. 2008. DOI: 10.1007/s12021-008-9024-z.

[29] D. N. Kennedy, C. Haselgrove, J. Riehl, N. Preuss and R. Buccigrossi, "The NITRC image repository.", *NeuroImage*, vol. 124, Part B, pp. 1069–1073, Jan. 2016. DOI: 10.1016/j.neuroimage.2015.05.074.

[30] H. Markram, "Seven challenges for neuroscience", *Functional Neurology*, vol. 28, no. 3, pp. 145–151, 2013. DOI: 10.11138/FNeur/2013.28.3.145.

[31] M. Feldman. (20th Jun. 2013). ''BigBrain" project makes terabyte map of a human brain, [Online]. Available: http://spectrum.ieee.org/tech-talk/biomedical/imaging/bigbrain-project-makes-terabyte-map-of-a-human-brain (visited on 20th Nov. 2015).

[32] G. A. Ascoli, D. E. Donohue and M. Halavi, "NeuroMorpho.Org: A central resource for neuronal morphologies", *Journal of Neuroscience*, vol. 27, no. 35, pp. 9247–9251, 2007. DOI: 10.1523/JNEUROSCI.2055-07.2007.

[33] M. Helmstaedter, K. L. Briggman and W. Denk, "High-accuracy neurite reconstruction for high-throughput neuroanatomy", *Nature Neuroscience*, vol. 14, no. 8, pp. 1081–1088, 2011. DOI: 10.1038/nn.2868.

[34] M. Helmstaedter and P. P. Mitra, "Computational methods and challenges for large-scale circuit mapping", *Current Opinion in Neurobiology*, vol. 22, no. 1, pp. 162–169, 2012. DOI: 10.1016/j.conb.2011.11.010.

[35] V. Marx, "Neuroscience waves to the crowd", *Nature Methods*, vol. 10, no. 11, pp. 1069–1074, 2013. DOI: 10.1038/nmeth.2695.

[36] Hanchuan Peng Group. (30th Mar. 2015). Vaa3D: Open-source, multi-dimensional data visualization and analysis, [Online]. Available: http://www.vaa3d.org/ (visited on 7th Aug. 2015).

[37] H. Peng, Z. Ruan, F. Long, J. H. Simpson and E. W. Myers, "V3D enables real-time 3D visualization and quantitative analysis of large-scale biological image data sets", *Nature Biotechnology*, vol. 28, no. 4, pp. 348–353, Mar. 2010. DOI: 10.1038/nbt.1612.

[38] H. Peng, A. Bria, Z. Zhou, G. Iannello and F. Long, "Extensible visualization and analysis for multidimensional images using Vaa3D", *Nature Protocols*, vol. 9, no. 1, pp. 193–208, Jan. 2014. DOI: 10.1038/nprot.2014.011.

[39] Y. Liu, "The DIADEM and beyond", *Neuroinformatics*, vol. 9, no. 2-3, pp. 99–102, Mar. 2011. DOI: 10.1007/s12021-011-9102-5.

[40] K. M. Brown, G. Barrionuevo, A. J. Canty, V. De Paola, J. a. Hirsch, G. S. X. E. Jefferis, J. Lu, M. Snippe, I. Sugihara and G. a. Ascoli, “The DIADEM data sets: Representative light microscopy images of neuronal morphology to advance automation of digital reconstructions”, *Neuroinformatics*, vol. 9, no. 2-3, pp. 143–57, Sep. 2011. DOI: 10.1007/s12021-010-9095-5.

[41] T. A. Gillette, K. M. Brown and G. A. Ascoli, “The DIADEM metric: Comparing multiple reconstructions of the same neuron”, *Neuroinformatics*, vol. 9, no. 2-3, pp. 233–245, Sep. 2011. DOI: 10.1007/s12021-011-9117-y.

[42] H. Peng, M. Hawrylycz, J. Roskams, S. Hill, N. Spruston, E. Meijering and G. A. Ascoli, “BigNeuron: Large-scale 3D neuron reconstruction from optical microscopy images”, *Neuron*, vol. 87, no. 2, pp. 252–256, Jul. 2015. DOI: 10.1016/j.neuron.2015.06.036.

[43] H. Peng, E. Meijering and G. A. Ascoli, “From DIADEM to BigNeuron”, *Neuroinformatics*, vol. 13, no. 3, pp. 259–260, Apr. 2015. DOI: 10.1007/s12021-015-9270-9.

[44] BigNeuron Consortium. (13th Mar. 2015). Frequently asked questions, [Online]. Available: https://alleninstitute.org/bigneuron/faq/ (visited on 7th Aug. 2015).

[45] Justice Management Division. (Jan. 2003). The Department of Justice systems development life cycle guidance document, [Online]. Available: http://www.justice.gov/archive/jmd/irm/lifecycle/table.htm (visited on 10th Nov. 2015).

[46] C. A. Schneider, W. S. Rasband and K. W. Eliceiri, “NIH Image to ImageJ: 25 years of image analysis”, *Nature Methods*, vol. 9, no. 7, pp. 671–675, 2012. DOI: 10.1038/nmeth.2089.

[47] M. Halavi, K. A. Hamilton, R. Parekh and G. A. Ascoli, “Digital reconstructions of neuronal morphology: Three decades of research trends”, *Frontiers in Neuroscience*, vol. 6, 2012. DOI: 10.3389/fnins.2012.00049.

[48] E. Meijering, “Neuron tracing in perspective”, *Cytometry Part A*, vol. 77A, no. 7, pp. 693–704, Mar. 2010. DOI: 10.1002/cyto.a.20895.

[49] R. C. Cannon, D. A. Turner, G. K. Pyapali and H. V. Wheal, “An on-line archive of reconstructed hippocampal neurons”, *Journal of Neuroscience Methods*, vol. 84, no. 1, pp. 49–54, 1998. DOI: 10.1016/S0165-0270(98)00091-0.

[50] C. Bauer, T. Pock, E. Sorantin, H. Bischof and R. Beichel, "Segmentation of interwoven 3D tubular tree structures utilizing shape priors and graph cuts", *Medical Image Analysis*, vol. 14, no. 2, pp. 172–184, 2010.

[51] D. Jiménez, M. Papadakis, D. Labate and I. A. Kakadiaris, "Improved automatic centerline tracing for dendritic structures", in *Proc. IEEE* 10th *International Symposium on Biomedical Imaging*, 2013, pp. 1050–1053. DOI: 10.1109/ISBI.2013.6556658.

[52] P. Hernandez-Herrera, M. Papadakis and I. A. Kakadiaris, "Segmentation of neurons based on one-class classification", in *Proc. IEEE* 11th *International Symposium on Biomedical Imaging*, 2014, pp. 1316–1319.

[53] J. Xie, T. Zhao, T. Lee, E. Myers and H. Peng, "Automatic neuron tracing in volumetric microscopy images with anisotropic path searching", in *Proc. International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, 2010, pp. 472–479.

[54] ——, "Anisotropic path searching for automatic neuron reconstruction", *Medical Image Analysis*, vol. 15, no. 5, pp. 680–689, 2011.

[55] G. Luo, D. Sui, K. Wang and J. Chae, "Neuron anatomy structure reconstruction based on a sliding filter", *BMC Bioinformatics*, vol. 16, no. 1, p. 342, 2015. DOI: 10.1186/s12859-015-0780-0.

[56] S. Gulyanon, N. Sharifai, S. Bleykhman, E. Kelly, M. D. Kim, A. Chiba and G. Tsechpenakis, "Three-dimensional neurite tracing under globally varying contrast", in *Proc. IEEE* 12th *International Symposium on Biomedical Imaging*, 2015, pp. 875–879.

[57] T. A. Gillette, "Comparative topological analysis of neuronal arbors via sequence representation and alignment", Dissertation, George Mason University, 2015, p. 274.

[58] D. Mayerich, C. Bjornsson, J. Taylor and B. Roysam, "Metrics for comparing explicit representations of interconnected biological networks", in *Proc. IEEE Symposium on Biological Data Visualization*, 2011, pp. 79–86. DOI: 10.1109/BioVis.2011.6094051.

[59] D. Mayerich, C. Bjornsson, J. Taylor and B. Roysam, "NetMets: Software for quantifying and visualizing errors in biological network segmentation.", *BMC Bioinformatics*, vol. 13 Suppl 8, no. Suppl 8, S7, 2012. DOI: 10.1186/1471-2105-13-S8-S7.

[60] S. Du Toit. (10th Sep. 2014). Hourglass interfaces for C++ APIs, [Online]. Available: http://www.slideshare.net/StefanusDuToit/cpp-con-2014-hourglass-interfaces-for-c-apis (visited on 22nd Nov. 2015).

[61] D. L. Donoho, A. Maleki, I. U. Rahman, M. Shahram and V. Stodden, “Reproducible research in computational harmonic analysis”, *Computing in Science & Engineering*, vol. 11, no. 1, pp. 8–18, 2009. DOI: 10.1109/MCSE.2009.15.

[62] C. Collberg, T. Proebsting and A. M. Warren, “Repeatability and benefaction in computer systems research”, University of Arizona, Technical Report TR 14-04, 27th Feb. 2015.

[63] B. Hook, *Write Portable Code: An Introduction to Developing Software for Multiple Platforms*. No Starch Press, Jul. 2005.

[64] U. Schwarz. (20th Jun. 2010). fdep: A pedestrian function dependencies finder, [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/17291-fdep--a-pedestrian-function-dependencies-finder (visited on 8th Jul. 2015).

[65] E. R. Gansner and S. C. North, “An open graph visualization system and its applications to software engineering”, *Softw. Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, Sep. 2000. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.3.CO;2-E.

[66] U. Drepper. (21st Nov. 2007). What every programmer should know about memory, [Online]. Available: http://www.akkadia.org/drepper/cpumemory.pdf (visited on 30th Oct. 2015).

[67] S. Savitzky. (1981). The programmer’s alphabet, [Online]. Available: http://steve.savitzky.net/Songs/alphabet/ (visited on 20th Nov. 2015).

[68] D. Goldberg, “What every computer scientist should know about floating-point arithmetic”, *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. DOI: 10.1145/103162.103163.

[69] W. Kahan, “Pracniques: Further remarks on reducing truncation errors”, *Commun. ACM*, vol. 8, no. 1, pp. 40–, Jan. 1965. DOI: 10.1145/363707.363723.

[70] L. Ibáñez and B. King, in *The Architecture of Open Source Applications: Structure, Scale, and a Few More Fearless Hacks*, A. Brown and G. Wilson, Eds., vol. 2, Kristian Hermansen, 2012, ch. ITK.

[71] M. Borgerding. (29th Jul. 2012). Kiss FFT, [Online]. Available: http://kissfft.sourceforge.net/ (visited on 30th Oct. 2015).

[72] B. Boehm, “Software risk management”, English, in *ESEC ’89*, ser. Lecture Notes in Computer Science, C. Ghezzi and J. McDermid, Eds., vol. 387, Springer Berlin Heidelberg, 1989, pp. 1–19. DOI: 10.1007/3-540-51635-2_29.

[73] M. Frigo and S. G. Johnson. (19th Mar. 2007). benchFFT: FFT accuracy benchmark results, [Online]. Available: `http://www.fftw.org/accuracy/` (visited on 30th Oct. 2015).

[74] M. Tasche and H. Zeuner, "Worst and average case roundoff error analysis for FFT", *BIT Numerical Mathematics*, vol. 41, no. 3, pp. 563–581, DOI: `10.1023/A:1021923430250`.

[75] Y. O. Halchenko and M. Hanke, "Open is not enough. let's take the next step: An integrated, community-driven computing platform for neuroscience", *Frontiers in Neuroinformatics*, vol. 6, no. 22, 2012. DOI: `10.3389/fninf.2012.00022`.