1 Introduction

Scientists construct quantitative models to explain observations about natural systems in a coherent and rigorous manner. These models can be characterized by their scope and validity. The *scope* of a model is the set of observable quantities that the model can generate predictions about, and the *validity* of a model is the extent to which these predictions agree with empirical observations of those quantities. Today, as the number of models and the quantity of empirical data increases, scientists face a grand challenge: efficiently discovering models whose scope is of interest and then characterizing their validity against a continually growing body of available evidence.

A new model is typically proposed by publishing a description of how it works along with an argument justifying its utility to some targeted scientific community. This argument is made in words, supported by a handful of relevant equations and figures, and judged by peer review. Ideally, reviewers ensure that the model's predictions are consistent with the available and relevant data and compare each model against previously published models. This job is becoming increasingly difficult; although authors may refer to relevant experimental data and provide citations into the literature, these are generally incomplete and biased, in that authors often focus on data favorable to their models and compare them to implausibly simple hypotheses.

If modeling papers contained detailed comparisons to all related experiments and competing models, publications would become encyclopedic and their main points obscured. A strength of model publication today is its focused description of how a model works and its conceptual and technical advances. A weakness, however, is that evaluating the scope and validity of a model is intractable using a publication alone. Publications tell us **how** a model works but are less effective at comprehensively reporting **which** goals it achieves and **how well** it achieves them. This problem is exacerbated as more data is gathered following publication. Although model validity may change in light of this new data, there is no systematic process in biology today for re-evaluating existing models. Although new data and its most important theoretical implications propagate informally through a scientific community or appear in periodic reviews, the original publications – a resource of first resort for new scientists and onlookers – are cited "as-is" in perpetuity.

We can distill the central problem discussed here as this: the process of scientific model validation today is not sufficiently rigorous, comprehensive, or ongoing. This hampers scientists' ability to make accurate predictions about experiments, compare models, and precisely identify outstanding research problems. To overcome these obstacles, we propose systematizing the model validation process by creating software and associated cyberinfrastructure dedicated to scientific model validation, supplementing today's system of peer-reviewed scientific publications to better support the modern shift toward science at a large scale.

1.1 Existing Efforts There are several well-developed facilities for data and model sharing in biology, but few if any facilitate evaluation of models against data directly. For example, the Collaborative Research In Computational Neuroscience (CRCNS) data-sharing website[1], and the Open Source Brain (OSB) repository[2] are facilities for data and model sharing in neuroscience, respectively. The CRCNS website hosts several excellent data sets of relevance to computational models; however, the data is not organized or annotated to indicate which models may explain that data, nor how well they do so. The OSB is a resource for standardized model descriptions that permit reliable execution. Models are published as-is, without any information about which specific data the published models aim to predict and how well they achieve their goals. We aim to bridge such resources, increasing the usefulness of each in turn.

There are related efforts in the machine learning community to develop models and validate them against standardized, publicly-available datasets according to well-specified metrics. Kaggle[3] drives model development by organizing competitions where training data is made public and competitors submit competing algorithms, compared automatically by cross-validated accuracy on an unrevealed test set. The success of Kaggle shows that open competitions are effective[4], and that a "modeling as a competition" paradigm attracts data scientists across traditional discipline boundaries. On the other hand, Kaggle focuses solely on a few general cases in machine learning: classification, regression, and clustering, where validation criteria are straightforward.

Discipline-specific competitions having a similar structure but with more sophisticated validation criteria within biology have also resulted in important advances. For example, the quantitative single neuron modeling competition (QSNMC)[5] addressed the complexity-accuracy tradeoff among reduced models of

excitable membranes; the "Hopfield" challenge[6] asked for neuronal network form, given function; the Neural Prediction Challenge sought the best stimulus reconstructions, given neuronal activity[7]; the Diadem challenge is advancing the art of neurite reconstruction[8]; examples from other subfields of biology abound[9]. Our challenge is to develop a general framework in support of distributed data-driven model validation workflows, in the flavor of these kinds of competitions, but where the validation criteria themselves are also specified collaboratively. Initially, we will focus on validation challenges in the biological sciences, particularly neuroscience.

2 Outcomes and Products

Our framework is organized around simple executable *validation tests* that compute agreement between a model prediction and an experimental observation. Model *scope* is identified with the set of tests that a model is capable of taking and *validity* is identified with how well the model performs on these tests.

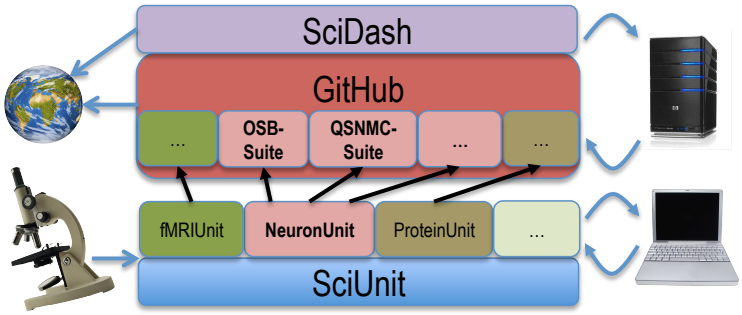

Figure 1: Proposal overview. i) The *SciUnit* framework can generate discipline-specific libraries for tests generation and model interfacing. *NeuronUnit* is proposed and described here. Experimental data guides test development; model generation informs and is informed by these *SciUnit* libraries. Each SciUnit test suite repository on GitHub is indexed by the *SciDash* web application. Test suites for OSB and QSNMC (described in the text) are being built. *SciDash* automatically runs the test collections in each suite and publicly displays the results.

This methodology is inspired by the ubiquitous practice of *unit testing* in software engineering. A *unit test* evaluates whether a portion of a computer program (often a single function) meets one simple correctness criterion. A *suite* of such tests *covering* the range of desired program behaviors helps validate the functionality of the program overall. Due to the narrow scope of each test, failed tests help precisely identify which parts of a program are functioning incorrectly. Developers often write unit tests before writing the program itself, following a methodology called test-driven development (TDD)[10]. When written early, tests serve as a partial program specification, guiding development and allowing developers to measure progress simply by looking at the proportion of tests passing at any point during development. When modifications are made, developers can ensure that *regressions* have not appeared by checking that all tests which passed before a change continue to pass afterwards. The success of unit testing and TDD in practice suggests that validation testing may be a practical foundation for model validation as well.

Inspired by these ideas, we propose here the development of ***SciUnit***, a framework for validation testing; ***SciDash***, a web application for collaboratively developing and discovering test suites, visualizing test results and organizing test-based competitions; and ***NeuronUnit***, a library of test-building and model-interfacing utilities for single-neuron neurophysiology that will serve as the most substantial initial case study for our tools. We will also facilitate the development, by collaborators, of test suites in other areas. At this time, we have interested collaborators in the field of fMRI-based brain imaging (eventually producing *fMRIUnit*). A diagrammatic overview of these pieces of our proposal is shown in Figure 1.

2.1 Example: Neural Membrane Potential Dynamics To make our proposal concrete, we will begin by describing a test suite related to single-neuron neurophysiology. The tests in this suite are designed for experiments where stimuli are delivered to neurons of a particular type, as somatically injected current (in pA), while the somatic membrane potential of each stimulated cell (in mV) is recorded and stored. A model claiming to capture this cell type's membrane potential dynamics must be able to accurately predict a variety of features observed in these data.

One simple validation test would ask candidate models to predict the number of action potentials generated in response to a series of stimuli, and compare these *spike count* predictions to the distribution observed in repeated experimental trials with the same stimuli. The test creator is responsible for specifying how to measure the goodness-of-fit; for example, a p-value derived from a chi-squared statistic could be calculated for each prediction and then these p-values could be combined using Fisher's method[11].

```
1  class SpikeCountTest(sciunit.Test):
2    """Tests models that generates spike counts from currents. Computes p-values based
         on a chi-squared test statistic, and pools them using Fisher's method.
3
4    parameters:
5      inputs: list of numpy arrays containing input currents (nA)
6      means, stds: list of means and standard deviations of the
7        spike counts observed in response to the inputs
8    """
9    def __init__(self, inputs, means, stds):
10     self.inputs, self.means, self.stds = inputs, means, stds
11
12   required_capabilities = [SpikeCountFromCurrent]
13
14   def run_test(self, model):
15     inputs, means, stds = self.inputs, self.means, self.stds
16     n = len(inputs)
17     counts = numpy.empty((n,))
18     for i in xrange(n):
19       counts[i] = model.spike_count_from_current(inputs[i])
20     chisquared = sum((counts-means)**2 / means) # An array of chi-squared values.
21     p = scipy.stats.chi2.cdf(chisquared,n-1) # An array of p-values.
22     pooled_p = sciunit.utils.fisherp(p_array) # A pooled p-value.
23     return sciunit.PValue(pooled_p, related_data={
24       "inputs": input_current,
25       "obs_means": means,
26       "obs_stds": stds
27     })
```

Figure 2: A single neuron spike count test family implemented in *SciUnit*

Alongside this *spike count test*, a number of other tests capturing different features of the data could be specified to produce a more comprehensive suite. For data of this sort, the QSNMC defined 17 other validation criteria in addition to one based on the overall spike count, capturing features like spike latencies (SL), mean subthreshold voltage (SV), interspike intervals (ISI) and interspike minima (ISM) that can be extracted from the data[5]. They then defined a combined metric favoring models that broadly succeeded at meeting these criteria, to produce an overall ranking. Such combined criteria are simply validation tests that invoke other tests to produce a result.

Importantly, the validation criteria in each case are made explicit by the specification of a test suite, so modelers need not guess which criteria are being used to validate or invalidate their model. Validation criteria are subject to debate (indeed, the QSNMC criteria changed between 2007 and 2008 due to such debates), and scientists who wish to promote different criteria need only derive alternative test suites. In many cases, models require no modifications to take the new tests because the same type of model output is being requested.

2.2 Validation Testing with *SciUnit* The first product of this proposal is a simple validation testing framework to support implementing test suites like the one described above. This framework, called *SciUnit*, is written in Python [12] and uses Python's object-oriented facilities to simplify test construction (from data) and execution[13]. Python was chosen due to its widespread adoption across the quantitative sciences, its open availability and because mature interoperability layers interfacing with other major languages used within science, including C[14], R[15], and MATLAB[16], are available. The functionality we will describe can also be readily translated into any programming language with comparable facilities, however.

Fig. 2 shows how a scientist would implement the spike count test described in the preceding section using *SciUnit*. A *SciUnit* validation test is an instance of a Python class implementing the `sciunit.Test` interface. Here, we show a class `SpikeCountTest` taking three *parameters* in its constructor (in Python, the constructor is always named `__init__`; the parameters' meaning is documented on lines 5-8). For convenience, we also make use of functions provided by the popular NumPy[17] and SciPy[18] libraries,

```
class SpikeCountFromCurrent(sciunit.Capability):
  def spike_count_from_current(self, input):
    """Takes a numpy array containing current stimulus (in nA) and
    produces an integer spike count. Can be called multiple times."""
    raise NotImplementedError("Model does not implement capability.")
```

Figure 3: An example capability specifying a single required method (used by the test in Figure 2).

```
class TrainSpikeCountFromCurrent(sciunit.Capability):
  def train_with_currents(self, currents, counts):
    """Takes a list of numpy arrays containing current stimulus (in nA) and
    observed spike counts. Model parameters should be adjusted based on this
    training data."""
    raise NotImplementedError("Model does not implement capability.")
```

Figure 4: Another capability specifying a training protocol (not used by the test in Figure 2).

although these are not required by *SciUnit*. To create a *particular* spike count test, we instantiate this class with particular experimental observations. For example, given observations from hippocampal CA1 cells (not shown), we can instantiate a test as follows:

```
CA1_sc_test = SpikeCountTest(CA1_inputs, CA1_means, CA1_stds)
```

We emphasize the crucial distinction between the *class* `SpikeCountTest`, which defines a *parameterized family* of validation tests, and the particular *instance* `CA1_sc_test`, which is an individual validation test. As we will describe below, we expect communities to build repositories of such families capturing the criteria used in their field so that test generation for a particular system of interest will often consist simply of instantiating a previously-developed family with particular experimental parameters and data. For single-neuron tests like the `SpikeCountTest`, we are developing a library to facilitate development (Sec. 2.7).

Classes that implement the `sciunit.Test` interface must contain a `run_test` method that receives a candidate *model* as input and produces a *score* as output. To specify the interface between the test and the model (that is, to constrain the test's scope), the test author provides a list of *capabilities* in the `required_capabilities` attribute, seen on line 12 of Fig. 2. Capabilities are simply collections of methods that a test may need to invoke to receive relevant data, and are analogous to *interfaces* in e.g. Java [19]. In Python, they are written as classes with unimplemented members. The capability required by the test in Fig. 2 (used on line 19) is shown in Fig. 3. In *SciUnit*, we require that classes defining capabilities be tagged by inheriting from `sciunit.Capability`, for discovery by our infrastructure (Figs. 3,4).

Capabilities are *implemented* by models. A simple *family* of models implementing this capability is shown in Fig. 5. Models in this family produce a spike count by applying a linear transformation to the mean of the provided input current. The family is parameterized by a scale factor and offset, both scalars. To create a *particular* model, the modeler may choose particular parameter values, just as with test families:

```
CA1_linear_model_heuristic = LinearModel(3.0, 1.0)
```

Here, the parameters to the model were picked by the modeler heuristically, or based on externally-available knowledge. An alternative test design to the one in Fig. 2 would add a second required capability (Fig. 4) and a corresponding training phase to the `run_test` method. This design would be relevant only for those models for which parameters can be adjusted without human involvement. Whether to build a training phase into the test protocol is a choice left to each modeler. Although Fig. 2 does not include a training phase, if training data is available then models that nevertheless implement a training capability (like `LinearModel`) can simply be trained explicitly by calling the capability method just like any other Python method:

```
CA1_linear_model_fit = LinearModel()
CA1_linear_model_fit.train_with_currents(CA1_training_in, CA1_training_out)
```

In the test in Fig. 2, the `run_test` method simply calls the `spike_count_from_current` capability method to produce spike count predictions for each input current on line 18. There are many possibly statistical choices for deriving a test score. In the figure, the resulting spike count is compared to the empirical

```
class LinearModel(sciunit.Model, SpikeCountFromCurrent,
    TrainSpikeCountFromCurrent):
  def __init__(self, scale=None, offset=None):
    self.scale, self.offset = scale, offset

  def spike_count_from_current(self, input):
    return int(self.scale*numpy.mean(input) + self.offset)

  def train_with_currents(self, currents, counts):
    means = [numpy.mean(c) for c in currents]
    [self.offset, self.scale] = numpy.polyfit(means, counts, deg=1)
```

Figure 5: A simple model that returns a spike count by scaling the mean of the input by a fixed parameter.

mean to produce a chi-squared statistic, and ultimately a p-value. Alternatively, the array of standard deviations (line 10) could be used to derive a Z-score. In addition to the score itself, the returned score object also contains metadata of use to those wishing to examine the result in detail. In this test we saved the inputs and observed means and standard deviations alongside the score by using the `related_data` parameter. We discuss visualization of results in Secs. 3.2 and 4.3.1.

Finally, a test is executed against a model instance, using the `sciunit.judge` function:

```
score = sciunit.judge(CA1_sc_test, CA1_linear_model_heuristic)
```

The `sciunit.judge` function operates in three phases: **(1) Capability Checking**: The model is verified as capable of taking the test by checking each capability in the test's `required_capabilities` attribute; **(2) Test Execution**: The test's `run_test` method is called to execute the model and the output is cached. **(3) Output Validation**: `run_test` returns an instance of a `sciunit.Score` subclass containing a goodness-of-fit metric between the model output and the data used to parameterize the test. Any errors that occur during this process are reported by raising an appropriate exception.

2.3 *SciUnit* Test Suites A test suite is a collection of tests designed to validate a model against several mutually coherent requirements, i.e. various summaries of the data. The following is a test suite that could be used for a simplified version of the QSNMC:

```
CA1_suite = sciunit.TestSuite([CA1_sc_test, CA1_sl_test, CA1_sv_test, CA1_isi_test,
    CA1_ism_test])
```

When a test suite is executed against a model, it produces summary data that can be shown on the console or visualized by other tools, such as the web application described in Sec. 2.6.

2.4 *SciUnit* Design Considerations Conceptual and practical simplicity – the absence of "heavy-lifting" – is an essential requirement for participation by scientists. Our design does not require the storage and release of raw data in standardized formats. Instead, the salient aspects of the data relevant to model validation are abstracted behind a test definition. Similarly, implementation details of a model (e.g. its programming language) need not be public or standardized – the implementation must simply be wrapped to expose standardized capabilities. Each community can collaboratively specify capabilities capturing the high-level semantics of their preferred modeling formalisms. Existing modeling standards (e.g. NeuroML[20, 21]) can be leveraged to simplify this process. **Thus, the first product of this proposal is a validation testing framework called *SciUnit*, written in Python, and designed to make test construction (from data) and test-driven model execution exceptionally easy and flexible.** All figures shown here containing code blocks make use of this framework.

2.5 Community Workflow In software engineering, suites of unit tests are produced by the community of developers responsible for a program. Analagously, a scientific community's members will collaboratively produce suites of validation tests in common source code repositories. Each test specifies a single requirement that a model must fulfill by specifying required capabilities and providing a procedure for determining whether a candidate model (the input to the test) is consistent with a single feature of the experimental data. Model performance on a collaboratively curated suite of validation tests can thus serve as

Table 1: Glossary of Terms

Data	The measurements yielded by an experiment.
Model	A set of relations that aims to explain, reproduce, or predict data. Realized by a class that generates a set of related, parameterized model instances. Each instance of one model differs only in the set of parameters (corresponding to experimental design or training data) with which it was initialized.
Validation	The process of assessing the agreement between model output and experimental data.
(Validation) Test	A set of criteria to evaluate a model's output against a summary of experimental data. Realized in a test instance. One experiment can inform the development of many tests.
Test Family	A class generating a set of related test instances. Each instance from one family differs only quantitatively, according to its initialization data. All tests from one family try to validate the same model property.
Test Suite	A collection of tests, each of which attempts to validate a different property of a model.
Score	A value summarizing the performance of a model on a test. This can be a measure of the goodness of fit between model output and experimental data.
Scope	A list of tests that a model is eligible to take, indicating the range of experimental data that the model might be able to explain or replicate.
Capability	A contract stating that a model accepts input or produces output of a particular form. A Hodgkin-Huxley model is capable of producing a membrane potential, whereas a model without a notion of voltage is not.
Record	An indication of the test score achieved by a model, along with any initialization conditions.
Record Suite	A list of records associated with a model. This summarizes the scope and validity of the model.

justification of claims regarding the model's validity as a description of the scientific system characterized by the test suite. This workflow continuously produces a summary of the field, indicating which models are capable of taking each test, i.e. whether the test falls within the model's scope; and for each capable model, how well it performed, i.e. its validity. Model usefulness becomes a function of the weight given to each test, determined per investigator or by community consensus. This test-driven scientific workflow leverages the desire of modelers to promote their models' virtues – test suites represent clear *challenges*, issued by experimentalists to the community, and passed tests certify success for the modelers. As more data is gathered and test suites are refined, past models can be tested continuously againt current empirical knowledge in a fully-automated manner, because the interface between tests and models is fixed.

2.6 *SciDash*: A Community web application This framework is most effective when the current state of a research area is represented by a collection of tests and models. This requires coordination among research groups, so community-oriented cyberinfrastructure to support test suite creation and summarization, building upon *SciUnit*, is essential. We propose a service, called *SciDash*, utilizing existing infrastructure for coordinated development of software repositories, focusing on GitHub[22][23]. A suite of related tests will be contained in a GitHub repository with a stereotyped high-level structure allowing the *SciDash* web application to discover, index and summarize it continuously (Fig. 6). ¡¡¡¡¡¡¡ HEAD

A web application written using the Django framework[24] will serve as a central location where scientists can discover relevant test suites, determine test requirements, and summarize the results of test execution on submitted models. Test results can be visualized as a "record matrix" composed of large numbers of model/test combinations (Fig. 7). Each row in this matrix will contain results for all tests taken by one model and would serve as clear evidence of that model's scope and validity. Models, tests, and records will be stored on GitHub, pointed to by hyperlinks in the record matrix. *SciDash* will serve as a public record of competition among models, facilitating and encouraging data-driven development among modelers.**Thus, the second product of this proposal is a web application called *SciDash*, whose back-end can index test suite repositories collaboratively developed on GitHub, and whose**

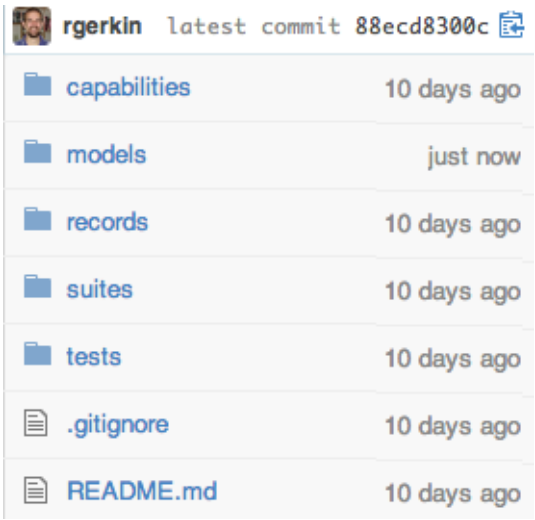

Figure 6: A *SciDash* repository on GitHub

Results for suite Single Neuron Suite B

Notify me of updates

Model	Submitter(s)	Overall ▼	SC	SL	SV	ISI	ISM
ARX	Shinomoto, Kobayashi	.95	.91	.96	.95	.90	.98
AdEx-1	Badel	.91	.95	.86	.91	.92	.94
aSRM	Mensi	.77	.85	.93	.71	.83	.44
Point Process	Kass	.56	.87	.73	N/A	.71	N/A

No Comments. Add a comment.

Figure 7: A *SciDash* record matrix

front-end offers users a simple environment for discovery and evaluation of models, tests, and records.

2.7 *NeuronUnit*: A Suite of Tests for Neurophysiology To demonstrate the utility of these tools, an initial case study must be conducted. Because we have substantial experimental and computational neurophysiology training and expertise, we target these domains initially. By using machine-readable models from Open Source Brain (OSB), and machine-readable data from resources like The NeuroElectro Project[25], our efforts will produce a body of useful tests that also serve as a demonstration of the test-driven workflow that other areas in the biological sciences can leverage. **The third product of this proposal is *NeuronUnit*, a large suite of data-driven tests, tools for capability implementation, and model interfaces for a representative set of canonical neurophysiology models, all publicly available. A case study examining the implementation of *NeuronUnit* will inform development of domain-specific test suites in other biology sub-fields.**

3 Preliminary Activities

3.1 *SciUnit* Activities We have developed all core features of the validation testing framework, *SciUnit*, under a free, open-source license at the project development repository[26]. Table 1 provides definitions of terms used in this proposal and in the code. *SciUnit* underlies the examples given in all code blocks here.

3.2 *SciDash* Activities We have also begun work on the *SciDash* web application, which points to test suite repositories and aggregates test results for scientific communities. A development version written using Django (a Python web framework) is currently live[27]. It is populated with models and tests from the educational illustration in Table 2 (section 5.1). We have implemented *SciDash* in three layers. The **first layer** is completely under the control of individual developer communities, and consists of git repositories. Git is the most popular implementation of "version control" among developers today[23], and is increasingly used for open science projects (e.g. OSB[2]). The GitHub website[22] is the most popular location for collaborative code development using git. It organizes millions of git repositories, provides numerous tools for collaboration and discussion, and easy-to-use GUIs for every operating system. Git and GitHub also meet all the requirements of a means for collaboration on test suite repositories: they are open (but provide user access controls); the entire history of code changes is visible; and collaboration is simple. For example, anyone can "fork" a repository on GitHub, make desired changes, and optionally submit those changes back to the main repository for inclusion. These virtues make GitHub an ideal first layer for *SciDash*. We created a vanilla test suite repository[28] on SciDash (Fig. 6). We are currently developing an API for *SciDash* to identify all similarly structured repositories (by tracking the lineage of the repository above using the GitHub API). This list of repositories will be indexed by *SciDash* to form the **second layer**, an overview of the state of all *SciDash* repositories on GitHub. This index will be searchable and filterable on the *SciDash* website to identify repositories of interest to a particular research community. The **third layer** will be a "dashboard" view of the state of a field, visible through a matrix of records for model/test combinations in a GitHub repository (Fig. 6). Each record will link to test records (stored in the repository), displaying line-by-line the executed code and intermediate output statements, as well as the underlying models and tests.

```
class CA1PyramidalCellModel(NeuroConstructModel):
    """CA1 Pyramidal Cell model from Open Source Brain."""
    def __init__(self,**kwargs):
        project_path = neuroconstruct.get_path("hippocampus","CA1_pyramidal_neuron",
            "CA1PyramidalCell","neuroConstruct")
        models.NeuroConstructModel.__init__(self,project_path,**kwargs)
```

Figure 8: A model class corresponding to a CA1 Pyramidal Cell model from Open Source Brain

3.3 *NeuronUnit* Activities Here we describe standard neuroinformatics tools we have adopted to develop *NeuronUnit*[29].

3.3.1 Models from NeuroML NeuroML is a standardized model description language for neuroscience [21]. It permits many neurophysiological/neuroanatomical models to be described in a simulator-independent fashion, and executed across many popular simulators due to inter-conversion capabilities of the NeuroML API. Because NeuroML is an XML specification, model descriptions can be validated for correctness and queried for model properties and components, exposing potential capabilities. It is ideal for model sharing, curation, and for answering both *what* and *how* programatically.

NeuroConstruct[30, 31] is a simulation manager that takes NeuroML models and hands off simulation to supported simulators. *NeuronUnit* offers a `sciunit.Model` subclass called `NeuroConstructModel`, instantiated with the path to a NeuroML model. Because NeuroML can describe such a wide range of models, `NeuroConstructModel` makes few assumptions about them: that each one is `TimeIntegrable`, and `HasMembranePotential`. It is subclassed to test *specific* NeuroML models (Fig. 8).

The Open Source Brain project (OSB,[2]) curates many models described in NeuroML. OSB-curated projects are converted from their native format into NeuroML, and run on major neural simulators[32–35]. Concordance between model output (beginning with the NeuroML description) and reference output (from native simulator source files) is reported for each model. Thus, OSB is an excellent source of models that, in addition to being open source, are sufficiently described to enable validation. The hippocampal CA1 pyramidal cell is commonly modeled, and we implement one such model hosted on OSB[36] by simply declaring a `CA1PyramidalCellModel` class, inheriting from `NeuroConstructModel`. This basic implementation simply "wraps" the components of the existing model, with simulator interaction taken care of by `NeuroConstructModel` methods; thus, only the code shown in Fig. 8 is required. All OSB models, and indeed any NeuroML model, can be tested similarly. Working together with OSB is part of our **first collaboration**, and our integration efforts can be publicly tracked[37].

Spanning a range of scales and original development environments, all OSB models are formally described using NeuroML, as are all model components and sub-components, such as cells, ion channels, calcium stores, etc. These models are regularly executed on OSB servers to ensure that their output remains consistent as they are updated. Therefore, OSB can confirm that they *do* work, while linked journal articles, on-site wiki, and code inspection can establish *how* they work. However, there is no mechanism for establishing *how well* they work, i.e. how well the models accord with data. *SciUnit* fills this gap by helping OSB (and the larger biology community) assess models using data-driven derived from the *NeuronUnit* library. *SciUnit* can be applied similarly to other biology sub-disciplines using *NeuronUnit* analogues written by the corresponding communities.

3.3.2 Capabilities from NeuroTools NeuroTools[38] is a Python library supporting tasks associated with analysis of neural data (or model output), such as membrane potential time series, spike trains, etc. It is an open source and actively developed project, containing reliable algorithms on which to base neurophysiology tests.

We use NeuroTools to implement *SciUnit* capabilities in *NeuronUnit*. For example, an `AnalogSignal` object (e.g. a membrane potential time series) from NeuroTools has a threshold detection method that returns a NeuroTools `SpikeTrain` object. A *NeuronUnit* `HasSpikeTrain` Capability requires that the method `getSpikeTrain` be implemented. `NeuroConstructModel` does so by placing the object method `AnalogSignal.threshold_detection` inside `getSpikeTrain`. Many such NeuroTools objects are similarly exchanged between `NeuroConstructModel` methods. This simplifies test writing, since basic model output properties are obtained trivially using NeuroTools object methods, and these NeuroTools ob-

```
# Interface with neuroelectro.org to search for spike widths of CA1 Pyramidal cells.
reference_data = NeuroElectroSummary(neuron={'id':85}, ephysprop={'id':23})
reference_data.get_values()  # Get summary data for the above.
model = CA1PyramidalCellModel(population_name="CG_CML_0") # Initialize the model
    with some parameters.
test = SpikeWidthTestDynamic( # Initialize the test.
    reference_data = {'mean':reference_data.mean, 'std':reference_data.std}, #
        Summary statistics from the reference data
    model_args = {'current':40.0}, # Somatic current injection in pA.
    comparator = ZComparator), # A comparison class that implements a Z-score.
result = sciunit.judge(test,model) # (1) Check capabilities, (2) take the test, (3)
    generate a score and validate it, (4) bind the score to model/test combination.
result.summarize() # Summarize the result.
```

Figure 9: Working example of a testing in *NeuronUnit*

jects are easily extracted from model output using candidate models subclassing `NeuroConstructModel`.

3.3.3 Reference Data for Tests from NeuroElectro Answering *how well* requires validation testing against data. The NeuroElectro project[25] is an effort to curate all published single cell neurophysiology data[39]. Currently, up to 27 electrophysiological properties are reported for 93 cell types, spanning > 2000 single pieces of published data extracted from article tables. We have made it easy to construct *NeuronUnit* tests using the NeuroElectro API to get reference data. Tests can be based upon data from single journal articles, or from ensembles of articles with a common theme (e.g. about a particular neuron type). The former is illustrated in Figure 9. Associated statistics of that data (e.g mean, standard error, and sample size) are attached and enable judgement of model output according to a chosen scoring mechanism. While NeuroElectro alone cannot judge all model aspects, it can serve to validate basic features of many neurophysiology models, such as resting membrane potential, action potential width, after-hyperpolarization amplitude, etc. As NeuroElectro is the only publicly curated source of such data, it represents a key component for *NeuronUnit* test constuction. Continued development of the NeuroElectro API, through which data are systematically exposed to test authors, represents our **second collaboration**[40].

4 Research and Development Plan

In order to make immediate use of *SciUnit* and populate *SciDash* with models and tests, we will initially focus on *NeuronUnit* development, to serve one discipline (neurophysiology) without compromising generalizability. This will enable rapid feedback from a familiar experimental community, guiding development.

4.1 A Complete Pipeline Although the tools described in Sec. 3.3 do not exhaust the possible sources of models, capabilities, and test data, they provide an immediate point of entry into the neurophysiology community and a powerful demonstration of our proposal. In the *NeuronUnit* repository[29] is a runnable script (*examples.py*) demonstrating a complete testing pipeline. It (1) selects an OSB model; (2) simulates it using NeuroConstruct; (3) tests the widths of the resulting action potentials, extracted and computed using NeuroTools, against NeuroElectro data downloaded on-the-fly, using a *NeuronUnit* test class called `SpikeWidthTestDynamic`; and (4) computes and prints a test score (Figure 9).

4.2 Creating New Models, Capabilities, and Tests *NeuronUnit* provides base classes to enable rapid generation of models, capabilities, and tests for neurophysiology data. However these objects can also be created from scratch, requiring only adherence to the *SciUnit* interface. For example, a Model could implement an `integrate` capability method by wrapping execution of a MATLAB script and a `get_spikes` capability method by parsing a .csv file on disk; a Test could be initialized using empirical spike rates collected in the lab. While this does not meet our idealized vision of development and testing, in practice this may be a common scenario. As part of our outreach efforts (Sec. 5) we will train modelers in the use of this framework, and encourage them to use it as part of their workflow of choice. Adoption of *SciUnit* into traditional, custom workflows would be reflected in the Methods sections of articles.

4.3 *SciDash* Implementation *SciDash* will point to collections of models and tests as they become available, and will display corresponding test results. This will provide a dedicated resource for evaluating model validity. The progress of the initiative, including community adoption, will be transparent from the number and size of the repositories indexed. The source code is being developed openly[41], supporting organizations that to host a private dashboard for models or datasets not yet ready for release. Organizations can set their own root *SciDash* repository, indexing only those repositories forked from that root.

4.3.1 Visualization There are many possibilities for visualization of specific test results though *SciDash*. When a test record is selected from the record matrix (Fig. 7), it can point to a Sage notebook[42], whose "worksheets" facilitate visualization and storage of executed code. While Sage was written for Python, the notebook also supports the execution of code and visualization of outputs from other programming languages, making possible worksheets based on MATLAB, Mathematica, and other popular environments for modeling. Even with *SciUnit* written in Python, calls to other languages can still be issued, and their results visualized using these worksheets.

4.3.2 Collaboration Community-moderated comments on GitHub will allow test-makers and test-takers to discuss issues associated with test suites. On GitHub, these takes the form of "issues," commit messages, and comments surrounding merge requests from forked repositories. Thus, disagreements about the appropriateness of a test can be openly aired and in many cases resolved. The *SciDash* website itself can also support public comment to extend the features of GitHub. More importantly, we will enable sorting and filtering of *SciDash* results by repository statistics, e.g. the volume of activity and the number of followers, via the GitHub API. This will allow the most important test suites, as judged by each community, to be featured prominently on *SciDash*. Simple tagging should enable filtering by subject matter. We also will support open authentication via existing web applications (Google, Twitter, etc.), lowering the barrier to participation.

4.3.3 Computation Initially, computations can be performed centrally on our hardware (high performance computer clusters), but as the load increases it will be important to encourage testing on contributor's machines. We support testing on any platform out-of-the-box via installation of *SciUnit*, a domain-specific library such as *NeuronUnit*, and the *SciDash* API to ensure that test records are properly organized for indexing by *SciDash*. Then visibility on *SciDash* is as simple as executing a single script that scans a stereotyped repository for tests and executes them. The *SciDash* backend will consist of a MySQL relational database, updated regularly using the GitHub API to populate record matrices.

4.3.4 Population *SciDash* will regularly run scripts that (1) read the GitHub repositories, (2) run any outstanding tests on new or existing models, and (3) format the results. In order to quickly populate *SciDash* with repositories that are both useful and illustrative, we will use *SciUnit* to recapitulate the results of some "settled" competitions as *SciDash* repositories, i.e. we will encode the competition rules as tests and *SciDash* will then index the results. There are several examples of settled competitions which have publicly available rules, entries, and results against which to check[5–7].

4.4 Measuring Progress Continuing with the tools described in Sec. 3.3 we will increase the number of *NeuroConstruct* capabilities implemented until adequate coverage of models on OSB is achieved. Progress will be tracked in an OSB *SciDash* repository. One measure of success will be construction of tests that parallel journal-published figures associated with OSB models. However, this will represent only a small subset of all possible model/test combinations. Another measure of success will be the number of commits, contributors, and forks (on GitHub) for this repository, which are viewable on all GitHub repository pages. An even more concrete measure of success would be for competitions to be run using *SciUnit*, with outcomes tracked on *SciDash*. To this end we will arrange to run the next QSNMC using *NeuronUnit*.

4.5 Project Milestones

Year 1: a) Refine the *SciUnit* core to complete the corresponding Python module; b) implement a suite of *NeuronUnit* tests using NeuroElectro data and capabilities using NeuroTools functions; c) Submit a manuscript describing the idea and tools.

Year 2: a) Automatie OSB model testing, greatly increasing the number and scope of models tested; b) Continue to define *SciUnit* model capabilities via collaborative development of NeuroTools; c) Ag-

gregate new community-provided datasets to generate *NeuronUnit* tests; d) Populate *SciDash* with a dozen repositories corresponding to focused test suites.

Year 3: a) Fully document *SciUnit* and associated tools to encourage adoption; b) Continue to write tests and specify (existing, published) models in NeuroML for testing; c) Submit a manuscript describing the overall results and promoting *SciDash*; d) Submit a manuscript describing the array of usable tests and tools in *NeuronUnit*; e) Promote use of these tools at conferences and workshops, generating community interest and inspiring development of *NeuronUnit* analogues in other biology disciplines.

4.6 Challenges Here we describe theoretical and practical challenges to implementation, and how they can be overcome.

4.6.1 Participation from Modeling Communities Modelers may not want to expose model capabilities, a requirement of test-taking. We anticipate four solutions: **First**, interfacing a model to *SciUnit* requires only implementing selected model capabilities. Often this means identifying native model procedures that satisfy a capability, and wrapping their functionality. This can require as little as one line of code. Importantly, the modeler is not required to expose or rewrite any model flow control. **Second**, we support multiple environments automatically by using NeuroML[21], and other simulator-independent model descriptions are possible for other domains. Automated generation of NeuroML from native model source code is in development (Gleeson, personal communication); for the popular NEURON simulator[32], this functionality is already mature and in use. This minimizes modeler effort for a large and growing number of models. **Third**, modelers have an incentive to demonstrate publicly their models' validity. Participation in public modeling competitions (Sec. 1.1) demonstrates this incentive. **Fourth**, modelers have an incentive to use *SciUnit* during development (see TDD, above) to ensure that ongoing development preserves correspondence between model and data. A popular test suite can represent a "gold standard" by which progress during development is judged.

4.6.2 Participation from Experimental Communities Experimentalists may not want to write tests derived from their data. We anticipate four solutions: **First**, tests require no special data formatting; only a list of required capabilities (for selecting eligible models), optional metadata (as run-time arguments), and a statistical data summary (for scoring tests) are required. A unit test is focused and does not require arbitrary computations on data. For example, suppose intracellular current injection evokes 100 action potentials, the width of which is of interest. Writing the test consists of selecting `ReceivesCurrent` and `ProducesActionPotentialShape` capabilities (one line of code each), computing the mean and variance of action potential widths (one line of code), specifying current injection parameters, e.g. amplitude and duration (two lines of code), and selecting a scoring mechanism from `sciunit.scores`, e.g. (colloquially) "Must be < 1 standard deviation of the mean" (one line of code). This example can be found in `NeuronUnit.tests.SpikeWidthTest`; heavy-lifting is done by the interface. **Second**, data-sharing is becoming accepted, and test-writing can be distributed across scientists, including non-experimentalists with other aims such as analysis or modeling. **Third**, many tests can be automatically generated using the NeuroElectro API, and the continued emergence of such data-aggregation initiatives will expand these possibilities. **Fourth**, an incentive to write tests for one's data exists: the ability to identify models that give the data clear context and impact.

4.6.3 Diversity of Levels and Kinds of Models and Data The diversity of topics in biology is vast. **First**, we address this by providing an interface allowing modelers to express specific capabilities. This capability set determines the range of eligible tests. Scale hierarchies are embedded in capability inheritance. For example, `HasActionPotentials` inherits from `HasNeurons`, and `HodgkinHuxley` inherits from `VoltageGated`. Thus, incompatibility of a test-requiring-action-potentials for a model-lacking-neurons is known without explicit tagging. **Second**, NeuroML naturally addresses diversity of scales because it is organized hierarchically, in "levels." Models can be sub- or supersets of other models; similarly for SBML[43, 44], a general systems biology markup language. **Third**, cross-level testing can use "Representional Similarity Analysis" (RSA)[45], requiring only that a model respond to defined inputs (e.g. stimuli). A "similarity matrix" for input responses defines a unique model signature, and can serve as intermediate test output. Goodness-of-fit between similarity matrices for model and experiment determines test scores;

these matrices are independent of model scale because their size depends only on test inputs, not system detail.

4.6.4 Appropriateness of Models for Validation Some models do not attempt to reproduce experiments, but serve as proof that some dynamical system has certain properties. Tests that abstract away experimental details can still inform and illuminate such models. Rather than encoding specific experimental stimulus and response values, a test could simply evaluate a mapping between sets of numbers. Some abstract models may have unexpected homology to that mapping, highlighting their relevance where it may otherwise have been missed. Alternatively, some models make specific experimental predictions, but require significant context not provided by a test. Rather than fail such models, they receive an "incomplete", i.e. no test record is generated for such models.

4.6.5 Arbitrary Scoring Criteria for Tests A test first assesses goodness-of-fit, and applies a normalization (e.g. pass/fail, 0.0-1.0) to generate a score. Arbitrary choices at both stages may benefit some models over others. **First**, however, rank-ordering is constant across many goodness-of-fit metrics, meaning that choice of metric will rarely cause an otherwise passing model to fail and vice versa. For example, given a data mean and variance, ordering model output by Z-score or p-value will yield the same relative ranking of models. Indeed, rank ordering of models may prove more valuable than test scores themselves. **Second**, suite repositories are open (e.g. Fig. 6), so tests can be cloned and new statistical choices implemented. Statistics as applied to neuroscience have been notoriously "fast and loose"; identification and correction of flawed methodology is becoming increasingly common[46–49], and is accelerated by an open framework. The community can judge which test version is most appropriate, i.e. what a model *should* do – this process documented via standard moderation techiniques used on GitHub – and the *SciUnit* framework determines whether the model *does* it.

4.6.6 Reliability of Data Underlying Tests Unreliable data can undermine model validation. **First**, the community must evaluate experimental design and methods, discounting data produced using questionable techniques. GitHub supports community moderation, permitting users to comment on tests, indicating their concerns. Suite repository popularity, by which *SciDash* results can be filtered, can reflect consensus. Experimental metadata also constrains a test's relevance, so test writers should select data with metadata appropriate to the system being modeled, and attach the metadata to resulting test scores. Metadata can also be expressed as Capabilities, e.g. `At37Degrees` or `Calcium3mM`; and tests can require that models express them. Such capabilities require no implementation, so the model definition must only inherit them. **Second**, models cannot perfectly reproduce data that is itself a random draw from a "true" distribution. Uncertainty in data must be made explicit, by asking how well a data set validates its own experimental replications[45]. The degree of such "self-validation" represents the upper limit of what a model can be expected to achieve, and should represent a "perfect" score.

4.6.7 Computational Efficiency Large models execute slowly, and repeated testing may be computationally intensive. We prioritize minimal re-execution of a model in *SciUnit*'s design. Test suites requiring the same model output many times require the model to be executed once, because model output is cached during test suite execution. This caching is implemented either as records of test workflow, e.g. Sage or IPython worksheets, or through optional database backends in the `sciunit.utils` module.

4.6.8 Occam's Razor All things being equal, simpler models are better. Model complexity has many definitions, so *SciDash* will report several complexity metrics[50], including: 1) model length; 2) memory use; 3) CPU load; 4) # of capabilities. *SciDash* will report the model validity vs complexity tradeoff in tabular form (e.g. Table 2), and in a scatter plot, with the "best" models being in the high validity / low complexity corner of the plot. The set of models which *dominate* all others, i.e. that have the highest validity for a given complexity, can be represented as a "frontier" in such a scatter plot, a visualization familiar from the symbolic regression package Eureqa[51].

4.6.9 Expansion Into Other Areas of Biology After covering neurophysiology, we would like *SciUnit* to be applied across neuroscience and in other biological sciences. The framework is discipline-agnostic, so community participation and model description are the only obstacles. Community participation begins

with enumerating the capabilities relevant to a sub-discipline, and then writing tests. Model description can expand within NeuroML (which already covers multiple levels within neuroscience) and tools for capability implementation can incorporate libraries for neuroimaging (NiBabel[52]), neuroanatomy (NeuroHDF,[53]) and other sub-disciplines. SBML[43, 44] will enable expansion beyond neuroscience, facilitated by parallel efforts among NeuroML developers to interface with it (Crook, unpublished). One intriguing possiblity is applying *SciUnit* to the OpenWorm project[54], which through open, collaborative development seeks to model the entire organism C. elegans.

5 Broader Impacts

5.1 Education The scientific method is a cornerstone of basic science education. However, its application in professional science is often informal and therefore of limited pedagogical value. *SciDash* will provide a window to the scientific method in practice, illustrating which hypotheses (models) withstand the scrutiny of evidence (pass tests). Hypothesis (model) revision will be transparent through version control and repository lineage on GitHub. The ability to visualize the scientific method in practice from any computer in the world will represent a major step forward in science education.

As a pedagogical tool, we will provide *SciDash* repositories for historical case studies. For example, Table 2 summarizes 5 astronomical models and 4 validation tests derived from, and recapitulating, the history of cosmology. This shows the scientific method to be an on-going process. Ptolemy's geocentric model[55] passes a test derived from Babylonian records of planetary motion. Ptolemy's model fails other tests shown here, including one constructed from Brahe's more meticulous measurements[56], falsifying Ptolemy's hypothesis of circular orbits. The Copernican heliocentric model[57] also uses circular orbits and fails Brahe's test, but is simpler than Ptolemy's model, dispensing with "epicycles." Kepler's laws of planetary motion permit elliptic orbits[58], as well as Galileo's discovery of elliptic orbits among Jupiter's moons[59]. Newton's model passes these tests and does so by unifying Kepler's laws under the principle of gravity[60]. Newton's model cannot explain Mercury's perihelion precession[61], solved only by General Relativity[62]. This account is simplistic compared with testing practice in modern biology, but it is appropriate for pre-baccalaureate teaching. *SciDash* will index and highlight such case studies for teaching.

Table 2: A record matrix illustrating models and data-driven tests from the history of cosmology. Each row is a model; each column is a test.

	Complexity	**Babylon**	**Brahe**	**Galileo**	**Le Verrier**
Ptolemy	Medium	Pass	Fail	Fail	Fail
Copernicus	Low	Pass	Fail	Fail	Fail
Kepler	Medium	Pass	Pass	Pass	Fail
Newton	Low	Pass	Pass	Pass	Fail
Einstein	High	Pass	Pass	Pass	Pass

These examples will also serve as an introduction to *SciUnit* specifically, which will help our second outreach effort: engaging students in model building and validation through focused competitions built on *SciUnit* and tracked using *SciDash*. Dr. Aldrich regularly teaches a computer programming class in which students are asked to create artificial lifeforms that must achieve specific objectives. He will update the curriculum to specify desired lifeform behavior using *SciUnit*-style tests, emphasizing to students the connection between modeling and the scientific method. Dr. Crook teaches a computational neuroscience class in which students build models of spiking neurons. Her students will use *NeuronUnit* to validate that the models achieve the desired output. In both cases, the use of the *SciUnit* framework during (and not simply after) development, i.e. test-driven development, will be encouraged. We believe that adoption of this framework will be most rapid when students learn to test their models as a habit and not an after-thought. To this end, Dr. Gerkin will sponsor a Google "Summer of Code" application[63] to fund interested high school students to improve upon existing models in areas of student interest using *SciUnit*. The sponsor ultimately selects from among the applicants, and we will actively seek minority and female applicants for this project to promote diversity.

5.2 Community Building *SciUnit* can make a significant impact on the field by enabling modelers and experimentalists to collaborate in building better models and in validating those models more rigorously. Modelers benefit by having a large set of readily-available experiments with which to refine and test their

models. Experimentalists benefit because their data will be more accessible to modelers, giving it more impact on the theoretical development of the field.

To that end, we plan a set of community-building activities that will educate modelers and experimentalists about the benefits of *SciUnit*, and teach them to use the related tools. We plan to organize a workshop, to be held at a popular computational neuroscience conference (e.g. CNS or Cosyne) focused on evaluation of scientific models; this workshop will help to raise awareness of the need for tools like *SciUnit*, and bring together a community well-suited to use it. The International Neuroinformatics Coordinating Facility (INCF, for which RCG serves on a data standards working group) also offers grants for "collaborations and knowledge transfer between and within INCF member countries", which could help fund workshops. We also plan to offer a tutorial on *SciUnit* at a summer Neuroinformatics course, the ideal choice being the one that RCG attended during his graduate training held at Marine Biological Labaratory. Finally, we will leverage *SciUnit* to organize a modeling competition, a sequel to the QSNMC[5], identifying modeling progress in neurophysiology.

5.3 External Outreach *SciUnit* can facilitate better understanding of science in the media and in the public at large. Today, the layperson has no reliable way to determine the importance or quality of scientific theories. Media reporters investigating new scientific results may consult scientists directly, but this approach suffers from bias and possible lack of available expertise. In constrast, *SciDash* will provide a less biased way for non-experts to identify model scope and validity, and compare these among competing models. With model and test sources well-documented in commit logs and code headers, contributors may be contacted for comment. Commentary embedded in GitHub repository issue tracking should also be a helpful resource for the media and the public.

SciDash's competitive nature will be appealing both to scientists and to the lay community. Competitions for machine learning, solar cars, and autonomous vehicles already draw considerable media coverage and biology (and the brain) is of no less interest to the public. Public competitions also welcome teams without academic credentials, such as students, to both learn the craft and possibly demonstrate key insights missed by professional scientists.

With projects open at every level, links embedded in social media can kindle public exploration of the models and tests that give these projects life. We will actively promote this framework by targeting media of interest to the relevant communities, via Twitter and blogs that increasingly cover "open" developments in scientific practice. Such avenues may have little import to senior scientists, but they are critical to communicating to the next generation's scientists, who have not yet established a go-to modeling workflow.

6 Personnel and Coordination

A scientific software framework succeeds by being used, which depends upon: **Relevance** to the needs of scientists; **Quality** of architecture and usability; **Conformance** to standards in targeted communities; **Integration** with existing software tools; **Applicability** to outstanding questions in a field; **Community** access (i.e. an accessible internet presence). We have the appropriate team members to meet these criteria.

6.1 [PIs] Richard C Gerkin, PhD (25 hrs/week) has expertise in experimental and computational neurophysiology, neuroinformatics, and web development. He will coordinate all project activities and be responsible for all output. He will also: coordinate with the NeuroElectro project and solicit other sources to obtain physiology data, and construct tests from data (experience as both an experimentalist and a modeler, across many scales, will facilitate this objective (**Relevance**)); write NeuroML-related bindings (e.g. `NeuroConstructModel`) (**Conformance**); subject models from OSB (and similar models publicly available) to tests; develop and maintain the *SciDash* website (**Community**); coordinate with OSB developers to implement automated *SciUnit* testing (**Integration**).

Sharon M Crook, PhD (2 hrs/week) has expertise in mathematics, computational modeling, and neuroinformatics. She will help interface *NeuronUnit* to NeuroML and OSB. **Community** access requires the use of an accepted model description standard. We chose NeuroML due to its advanced state and broad coverage of multiple scales (**Quality**). Dr. Crook is NIH-funded to maintain NeuroML and can help extend it if needed for model specification (**Conformance**). She will also assist with graduate training as needed.

Jonathan Aldrich, PhD (2 hrs/week) has expertise in software engineering, software verification and

validation, and human factors in software design. He will guide overall software architecture and supervise and train Mr. Omar. His expertise in software design for science and engineering applications will be key in this role, and assure **quality**. Dr. Aldrich will provide guidance on the overall structure of implementation. Dr. Aldrich is Mr. Omar's graduate supervisor.

6.2 [Key Personnel and Collaborators] Cyrus Omar (25 hrs/week) is a senior graduate student with expertise in computational modeling in neuroscience, computer science, and software infrastructure for science. He has been actively involved in *SciUnit*'s design, development, and software architecture (**Quality**) and will continue to maintain the core *SciUnit* Python module. He will get feedback from Dr. Gerkin about testing practicalities, informing code revisions.

R. Angus Silver, PhD is a full professor with expertise in neurophysiology, computational neuroscience and neuroinformatics. A frequent collaborator of Crook, Silver maintains OSB, funded by The Wellcome Trust. He has welcomed us to integrate *SciUnit* into OSB for automated model testing (**Integration**). This exposes a wide range of neuroscience models to testing (**Applicability**), and exposes the project to the international neuroscience community, serving as proof of concept for expansion into other areas of biology.

Shreejoy Tripathy, PhD is a recent doctoral awardee with expertise in neuroinformatics, computational neuroscience, and data mining. A frequent collaborator of Gerkin, Tripathy maintains The NeuroElectro Project (**Relevance**). He will help further **integrate** *NeuronUnit* with NeuroElectro through API development, providing a wide range of data for testing neuroscience models.

6.3 Means of Coordination Dr. Gerkin trained in Pittsburgh and maintains ongoing collaborations with Dr. Aldrich, Mr. Omar, and Dr. Tripathy. Dr. Gerkin works at ASU and regularly meets with Dr. Crook for "Math Biology" group meetings. Local, topical meetings, e.g. Statistical Analysis of Neural Data (SAND, in Pittsburgh yearly) represent another chance for the team to assemble and brainstorm in person. Because the existing projects underlying each collaboration are a) well-documented, and b) available under an open license, there are no barriers to code sharing and collaborative development. Github will be used for code development and communication between developers.

7 Current and Previous NSF Funding

Sharon Crook: [Intellectual Merit]: Sharon Crook was PI on NSF II-0613404, $457,654, CRCNS: Behaviorally Relevant Neuronal Modification during Postembryonic Development, funded from 10/06 through 9/10. This work examined how dendritic structure and synapse distributions follow functional architecture principles relating to a neuron's individual behavioral requirements in holometabolous insects, e.g. *Manduca sexta* and *Drosophila melanogaster.* Modeling and experimental activities focused on flight motoneurons, which transform from tonically firing larval crawling motoneurons into phasically firing adult flight motoneurons. This ongoing work has contributed to five publications[64–68] with two more manuscripts in review, seven published abstracts[69–75]. and 12 poster or oral presentations at conferences and workshops. **[Broader Impacts]:** This grant also funded training of four graduate students, including three women, and a Hispanic male postdoctoral fellow.

Jonathan Aldrich: [Intellectual Merit]: In his work on NSF CAREER grant CCF-0546550, "Lightweight Modeling and Enforcement of Architectural Behavior" (2006-2010, $500,000), Aldrich and his students developed a new approach to extracting run-time architectures from code and analyzing their behavior, producing 12 major conference[1] and journal publications and numerous workshop papers, including [76–84]. **[Broader Impacts]** The NSF grant mentioned above, and 3 others, have contributed to the education of 11 Ph.D. students, including 4 women, and at least that many undergraduate or master's students. The grants also supported development and release of several open-source tools, including ArchJava[85], Plural[86], Crystal[87], and SASyLF [88]. All of these tools, in addition to educational methodologies developed in the grants, have been used in education at Carnegie Mellon and other institutions. Recognition of the broader impacts of Aldrich's work on software architecture include the 2007 Dahl-Nygaard Junior Prize and an ICSE most influential paper award.

[1] note that in the field of computer science, major conferences are more prestigious than journals