Collaborative Infrastructure for Test-Driven Scientific Model Validation

Cyrus Omar, Jonathan Aldrich
Carnegie Mellon University
{comar,aldrich}@cs.cmu.edu

Richard C. Gerkin
Arizona State University
rgerkin@asu.edu

ABSTRACT

One of the pillars of the modern scientific method is *model validation*: comparing a scientific model's predictions against empirical observations. Today, a scientist demonstrates the validity of a model by making an argument in a paper and submitting it for peer review, a process comparable to *code review* in software engineering. While human review helps to ensure that contributions meet high-level goals, software engineers typically supplement this with *unit testing* to get a more complete picture of the status of a software project, particularly for complex projects involving many developers.

We argue that a similar test-driven methodology would be valuable to scientific communities as they seek to validate increasingly complex models against growing collections of empirical data. The dynamics of scientific communities and software communities differ in several key ways, however. In this paper, we introduce *SciUnit*, a framework for test-driven scientific model validation. We describe how SciUnit, supported by new and existing collaborative infrastructure, can integrate into the modern scientific process.

1. INTRODUCTION

Scientific theories often take the form of a *quantitative model*: a formal structure that can generate predictions about observable quantities. Such a model is characterized by its *scope*: the set of observable quantities that the model can predict, and by its *validity*: the extent to which its predictions agree with experimental observations of these quantities.

Quantitative models are validated by *peer review*. For a model to be accepted by the scientific community, scientists must submit a paper describing it and providing evidence that it predicts some quantity of interest more accurately than previous models, or that it makes a desirable tradeoff between accuracy and complexity [1]. Other members of the relevant community are then tasked with ensuring that validity was measured properly and that relevant data and competing models were adequately considered, drawing on knowledge of statistical methods and the prior literature. Publishing is a primary motivator for most scientists [2].

Quantitative scientific modeling and software development have much in common. Indeed, quantitative models are increasingly implemented as computer programs and in some cases, the program *is* the model (e.g. complex simulations). The peer review process for papers is similar in many ways to the *code review* process used in many development teams, where team members look for errors, enforce style and architectural guidelines and check that the code is *valid* (i.e. that it achieves its intended goal) before permitting it to be committed to the primary source code repository.

Code review can be quite effective [3], but this requires that developers expend considerable effort [4]. Code review is also most effective for resolving issues related to software architecture. Most development teams thus supplement code reviews with automated approaches to verification and validation, the most widely-used of which is *unit testing* [5]. In brief, unit tests are functions that check a single portion of a program against a single well-defined correctness criterion. A suite of such tests can be seen as a partial specification of a program or component. Test suites complement code review by allowing developers to answer questions like these more easily:

1. Which functionality has been adequately implemented? What remains to be done?
2. What modes of usage does the team consider most important? How does the team measure correctness?
3. Does a code contribution cause *regressions* in other parts of a program?

Scientists ask analagous questions:

1. Which observations are already well-explained by existing models? What is the state-of-the-art? What are the open modeling problems of interest?
2. What are the contemporary community standards for measuring goodness-of-fit?
3. How do newly-made experimental observations impact the validity of previously-published models?

But while software engineers can rely on a program's test suite, scientists today must extract this information from a body of scientific publications. This is increasingly difficult. Each publication focuses on just one model and is frozen in time, so it does not consider the latest experimental data

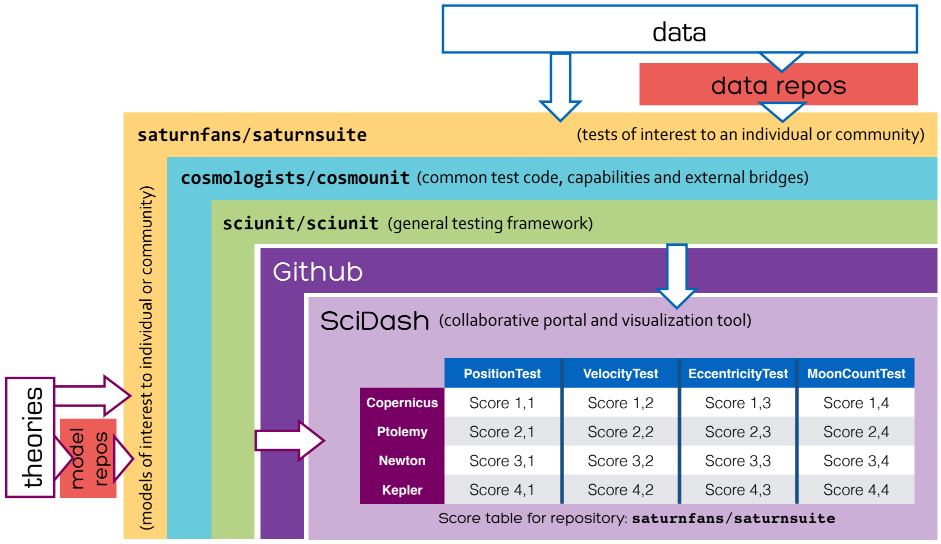

Figure 1: Tests are derived from data and models are derived from scientific theories. The score table summarizes the performance of a collection of models against a suite of tests. A table is contained in a *suite repository* (e.g. `saturnsuite`), hosted on social coding infrastructure – here, Github. SciDash is a portal that discovers and organizes these suite repositories and provides tools for visualizing them. Tests interface with models using capabilities, as specified by the testing framework (`sciunit`). Common testing code, capabilities and bridges to external model and data repositories are also collaboratively developed (e.g. `cosmounit`).

or statistical methods. Discovering, precisely characterizing and comparing models to discover the state of the art and find open modeling problems require an encyclopedic knowledge of the literature. Senior scientists can attempt to fill this need by publishing review papers, but in many areas, the number of publications generated every year can be overwhelming [6], and comprehensive reviews of a particular area are published relatively infrequently. Statisticians often complain that scientists are not following best practices and that community standards evolve too slowly because a canonical paper or popular review used outdated methods. Furthermore, if the literature simply doesn't address an important question of validity, a researcher often needs to reimplement a model from scratch.

One might compare this to a software development team relying primarily on carefully-prepared and cross-referenced summaries of code review sessions to understand, validate and evolve a complex software project where portions of the original source code have been obscured or lost. Although certainly a caricature, this motivates our suggestion that the scientific process could be improved by the adoption of test-driven methodologies alongside traditional peer review. However, the scientific community presents several challenges that must be addressed by any such methodology:

1. Unit tests are typically pass/fail, while goodness-of-fit between a model and data is typically measured by a continuous metric (e.g. a p-value).

2. Unit tests often test a *particular* component, whereas a *validation test* must be able to uniformly handle any model capable of predicting the quantity being tested.

3. Different modelers often prefer different programming languages or frameworks. Similarly, data formats can differ between experimentalists. The testing framework must be as flexible as possible about these choices.

4. Professional software developers are typically trained in testing practices and tools, while scientists rarely have training or experience with testing practices [7]. Thus, the framework must be as simple as possible.

5. Different communities, groups and individuals prefer different goodness-of-fit metrics and emphasize different sets of observable quantities. These can evolve as, for example, statisticians develop better measures. In contrast, there is more pressure to agree upon requirements and priorities in a software development project.

To address challenges 1-4, we will introduce a lightweight scientific validation testing framework, *SciUnit*, in Sec. 2. Challenge 5 has to do with coordination between scientists. To address this, we introduce a community workflow based on widely-adopted social coding tools (here Github) along with a lightweight community portal called *SciDash*. The overall goal of this work is to help scientists generate and examine tables like the one shown to Figure 1, where the relative validity of a set of models having a common scope can be determined by examining scores produced by a suite of validation tests constructed from experimental data.

2. VALIDATION TESTING WITH SCIUNIT

As a motivating example, we will begin by considering a community of early cosmologists recording and attempting to model observations of the planets visible in the sky, such as their position, velocity, orbital eccentricity and so on. One simple validation test might ask a model to predict planetary position on night $n+1$ given observations of its position on n previous nights. Figure 2 shows how to implement a test, using SciUnit, that captures this logic.

```
1  class PositionTest(sciunit.Test):
2      """Tests a planetary position model based on
           positions observed on day n given the
           positions in the n-1 previous days.
3      Metric: Standard p-value.
4      Parameters:
5        obs_histories : list[list[position]]
6        obs_positions: list[position]"""
7    def __init__(self, obs_histories, obs_positions):
8      self.obs_histories = obs_histories
9      self.obs_positions = obs_positions
10
11   required_capabilities = [PredictsPlanetaryPosition]
12
13   def _judge(self, model):
14     predictions = []
15     for obs_history in self.obs_histories:
16       predictions.append(model.predict_next_pos(
             obs_history))
17     p = pooled_p_val(predictions, self.obs_positions)
18     return sciunit.PValue(p, related_data={
19       'obs_histories': obs_histories,
20       'obs_positions': obs_positions,
21       'predictions': predictions
22     })
```

Figure 2: An example test class in `cosmounit`.

Before explaining the details of this example, we point out that SciUnit is implemented in Python. Python is one of the most widely used languages in science today [8]. It supports calling into many of the other popular languages, including R, MATLAB, C and Java, more cleanly than these languages support calling into Python (challenge 3). It is widely-recognized as being easy-to-read and its object system can be used to define abstract interfaces, which we leverage in support of challenge 2.

A *SciUnit* validation test is an instance of a Python class implementing the `sciunit.Test` abstract interface (line 1). Here, the class `PositionTest` takes two *parameters* in its constructor (constructors are named `__init__` in Python, lines 7-9). The meaning of each parameter along with a description of the goodness-of-fit metric used by the test is documented on lines 2-6. To create a *particular* position test, we instantiate this class with particular planetary observations. For example, the subset of cosmologists interested specifically in Saturn might instantiate a test by randomly chunking observations made about Saturn as follows:

```
h, p = randomly_chunk(saturn_obs_positions)
saturn_position_test = SpikeCountTest(h, p)
```

The class `PositionTest` defines logic that is not specific to any particular planet, so it is contained in a package shared by all cosmologists called `cosmounit`, while the particular test above would be used in a test suite focused specifically on Saturn called `saturnsuite`. Both the common logic and specific suites are collaboratively developed by these overlapping research communities in source code repositories on Github (or another similar service).

Classes that implement the `sciunit.Test` interface must contain a `_judge` method that receives a candidate *model* as input and produces a *score* as output. To specify the interface between the test and the model, the test author provides a list of *capabilities* in the `required_capabilities` attribute, seen on line 11 of Fig. 2. Capabilities are simply collections of methods that a test will need to invoke in order to receive relevant data, and are analogous to *interfaces* in e.g. Java. In Python, capabilities must be written as classes with unimplemented members. The capability required by the test in Figure 2 is shown in Figure 3. In *SciUnit*, classes

```
class PredictsPlanetaryPosition(sciunit.Capability):
  def predict_next_pos(self, history):
    """Takes a list of previous positions and produces
       the next position."""
    raise NotImplementedError("Model does not
        implement capability.")
```

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

defining capabilities are tagged as such by inheriting from `sciunit.Capability`. The test in Figure 2 repeatedly uses this capability on line 16 to produce a position prediction for each observation. We assume that positions are represented in a standardized manner specified within `cosmounit`. A model is simply an object that implements capabilities (via Python's simple inheritance mechanism).

The remainder of the `_judge` method compares the model predictions to observed data to produce a pooled p value. The method returns an instance of `sciunit.PValue`, a subclass of `sciunit.Score` that has been included with *SciUnit* due to its generality. In addition to the p-value itself, the returned score object also contains metadata, via the `related_data` parameter, for scientists who may wish to examine the result in more detail later. This illustrates some key differences between unit testing, which would simply produce a boolean result, and our conception of scientific validation testing (challenge 1). A score must induce an ordering, so that the table shown in Figure 1 can be sorted along its columns, and it can optionally specify a normalization scheme so the cells can be color-coded (not shown). A test can be manually executed using the `judge` method:

```
score = saturn_position_test.judge(kepler_sat)
```

This method proceeds by first checking that the provided model implements all required capabilities before calling the test's `_judge` method to produce a score. A reference to the test and model are added to the score for convenience (accessible via the `test` and `model` attributes, respectively).

Alongside this position test, we could also provide a number of other test classes in `cosmounit` and instantiate them with data about aspects of Saturn's motion (e.g. it's velocity) to produce a comprehensive suite in `saturnsuite`.

```
saturn_motion = sciunit.TestSuite([
      saturn_position_test, saturn_velocity_test, ...])
```

Like a single test, a test suite is capable of judging one or more models. The result is a score matrix much like the one diagramed in Fig. 1.

```
sm_matrix = saturn_motion.judge([copernicus_sat,
      ptolemy_sat, newton_sat, kepler_sat])
```

A test suite requires the union of the capabilities required by the tests it contains. A model that performs well across tests in such a suite could reasonably claim (e.g. to reviewers) that it is a coherent, valid model of Saturn's motion. As new data is collected, new tests can be added to the suite. However, because the interface between the test and the model remains the same (only the data parameterizing the tests changes), the table can be updated automatically. Similarly, when a new model is developed, it can immediately be evaluated against all known data that has been encoded as a test by simply exposing its predictions via the capabilities the test requires.

3. COLLABORATIVE WORKFLOW

The design just described has been purposefully left as simple as possible, in pursuit of challenge 4. Despite its simple design, it captures the essential elements of the scientific model validation process and satisfies the four challenges we laid out. SciUnit can be used by individual scientists to organize their workflows, but because science, and in particular, model validation is a collaborative process, we have also described the intended use of SciUnit within a collaborative workflow, mediated by a social coding tool like Github.

More directly: we anticipate common testing logic (e.g. `PositionTest`), modeling logic and capabilities being collaboratively developed by larger communities in less specialized repositories like `cosmounit`. Individuals and small groups will then parameterize these tests and models with data they have gathered, as well as data known from the literature and data contained in existing data repositories to create test suites in repositories like `saturnsuite`. The interface between data collection software and existing data representation standards and tests will be mediated by bridge logic also contained in repositories like `cosmounit`.

Statisticians wishing to promote new validation metrics can simply for a X`unit` repository and implement new test logic. By reusing the same interfaces as previous tests used, these new metrics can immediately be used to validate or invalidate a large body of existing models, providing valuable data for use in convincing the community to adopt these into the mainstream repositories. Similarly, investigators who wish to de-emphasize or emphasize different quantities can fork X`suite` repository and add or remove tests.

To help organize these repositories, a simple collaborative portal sitting above Github called *SciDash* is currently under development (*http://scidash.org/*). SciDash consists of an organized, collaboratively filtered listing of X`unit` and X`suite` repositories. That is, it is a tool to help scientists find the most popular repositories in their research area, to facilitate the development of community standards. SciDash also supports extracting summaries of tests, models, scores and related data from suite repositories to generate hyperlinked score tables and documentation. This facilitates exploratory workflows. To modify a suite (e.g. by adding the model a scientist is developing to it), a scientist can simply fork the repository. SciDash automatically discovers public forks of repositories that it is already indexing.

4. DISCUSSION

Academic peer review and code review are similar, but software developers typically supplement human review with test-driven methodologies. Such methodologies may benefit scientific communities as well, but several unique constraints make reusing existing testing frameworks and processes difficult. We describe a core testing framework that addresses these issues by using an existing, widely-adopted language and designing a flexible, simple framework that captures the domain-specific structure of scientific model validation. We then describe, by example, how the various components can be organized into software repositories and collaboratively maintained using social coding tools to support existing scientific practices. We end by outlining a collaboratively filtered portal that sits atop these tools and facilitates exploratory analyses and repository discovery.

While we discuss a toy example based on planetary movement here, we have applied this framework to more realistic problems in neurobiology and have bridged SciUnit to large-scale neuroinformatics projects (we anticipate publishing a description of these case studies shortly). Much future work remains to be done to investigate whether these tools are truly usable and useful to scientific communities, to further develop the community workflow and infrastructure and to develop a range of realistic case studies. Nevertheless, we believe that identifying the synergies between testing practices and model validation, and describing the basic tooling, represents a novel contribution to the literature on software testing. The core SciUnit framework has been fully developed and is available at http://sciunit.scidash.org/. SciDash is under active development, and is currently capable of basic forms of the operations described in Sec. 3.

Acknowledgements

We thank Sharon Crook, Shreejoy Tripathy and Padraig Gleeson for their support of this project. The work was supported in part by grant R01MH081905 from the National Institute of Mental Health. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

5. REFERENCES

[1] G. E. Box and N. R. Draper, *Empirical model-building and response surfaces.* John Wiley & Sons, 1987.

[2] J. Howison and J. Herbsleb, “Scientific software production: incentives and collaboration,” in *Proceedings of the ACM 2011 conference on Computer supported cooperative work.* ACM, 2011, pp. 513–522.

[3] H. Siy and L. Votta, “Does the modern code inspection have value?” in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, 2001, pp. 281–289.

[4] C. F. Kemerer and M. C. Paulk, “The impact of design and code reviews on software quality: An empirical study based on psp data,” *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 534–550, 2009.

[5] K. Beck, *Test Driven Development: By Example.* Addison Wesley, 2003.

[6] A. E. Jinha, “Article 50 million: an estimate of the number of scholarly articles in existence,” *Learned Publishing*, vol. 23, no. 3, pp. 258–263, Jul. 2010.

[7] J. Segal, “Models of scientific software development,” in *SECSE 08, First International Workshop on Software Engineering in Computational Science and Engineering, 13 May 2008, Leipzig, Germany.*, May 2008, Conference Item; PeerReviewed.

[8] M. F. Sanner *et al.*, “Python: a programming language for software integration and development,” *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.