# Collaborative Infrastructure for Test-Driven Scientific Model Validation

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

Richard C. Gerkin
Arizona State University, USA
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 it with *unit testing* to get a more complete picture of the status of a project, particularly when it is complex and involves many contributors.

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

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

## General Terms

Documentation, Measurement, Standardization

## Keywords

unit testing, model validation, cyberinfrastructure

## 1 INTRODUCTION

Scientific theories are increasingly being organized around *quantitative models*: formal systems capable of generating predictions about observable quantities. A model can be characterized by its *scope*: the set of observable quantities that it attempts to predict, and by its *validity*: the extent to which its predictions agree with experimental observations of these quantities.

Quantitative models are today validated by *peer review*. For a model to be accepted by a scientific community, its advocates must submit a paper that describes how it works and provides evidence that it makes more accurate predictions than previous models (or that it makes a desirable tradeoff between accuracy and complexity) [2]. Other members of the relevant community are then tasked with ensuring that validity was measured properly and that all 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 [3].

Quantitative scientific modeling shares much in common with software development. Indeed, quantitative models are increasingly being implemented in software and in some cases, the software *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 mistakes, 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 [9], but this requires that developers expend considerable effort [5]. Most large development teams thus supplement code reviews with more automated approaches to verification and validation, the most widely-used of which is *unit testing* [1]. In brief, unit tests are functions that check that the behavior of a single component satisfies a single functional criterion. A suite of such tests complements code review by making it easier to answer questions like these:

1. What functionality is a component expected to have?
2. What functionality has been adequately implemented? What remains to be done?
3. Does a candidate code contribution cause *regressions* in other parts of a program?

Scientists ask analagous questions:

1. What is a model's scope and how is validity measured?
2. Which observations are already explained by existing models? What are the best models of a particular quantity? What data has yet to be explained?
3. What affect do new observations have on the validity of previously published models? Can new models explain previously published data?

But while software engineers can rely on a program's test suite, scientists today must extract this information from a body of scientific papers. This is increasingly difficult. Each paper is narrowly focused, often considering just one model or dataset, and is frozen in time, so it does not consider the latest models, data or statistical methods. Discovering, precisely characterizing and comparing models to discover the state of the art and find open modeling problems can


*ICSE* '14, May 31 - June 7, 2014, Hyderabad, India

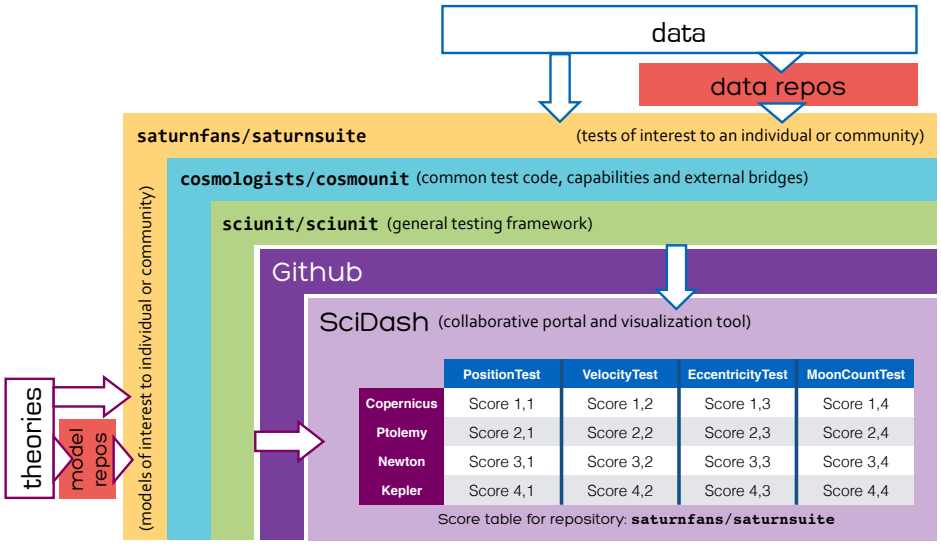


**Figure 1: Tests are derived from data and models are derived from scientific theories, either directly or through existing data and model repositories. The score table summarizes the performance of a collection of models against a suite of tests. Tests and models are initialized, executed and visualized using `sciunit` in an IPython notebook stored inside a *suite repository* (e.g. `saturnsuite`) hosted on a social coding website (e.g. Github). SciDash discovers and organizes suite repositories and provides interactive views of the notebooks they contain. Common testing code, capabilities and bridges to external model and data repositories are also collaboratively developed (e.g. `cosmounit`).**

require an encyclopedic knowledge of the literature, as can finding all data relevant to a model. Senior scientists often attempt to fill this need by publishing review papers, but in many areas, the number of publications generated every year can be overwhelming [4], and comprehensive reviews of a particular area are published 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 might need to reimplement an existing model in order to evaluate it.

One might compare this to a large software development team answering the questions listed above based largely on carefully reviewed, but rarely updated, API documentation and annual high-level project summaries. While 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 unique challenges:

1. Unit tests are typically pass/fail, while goodness-of-fit between a model and data is typically measured by a continuous metric.
2. Unit tests often test a *particular* component, whereas a *validation testing* system must be able to handle and compare many models with the same scope.
3. Scientists use a variety of programming languages.
4. Scientists often only loosely coordinate their efforts.
5. Different communities, groups and individuals prefer different goodness-of-fit metrics and focus on different sets of observable quantities. In contrast, there is more pressure to agree upon requirements and priorities in a typical software development project.
6. Professional software developers are typically trained in testing practices and the use of collaborative tools, while scientists more rarely have such experience [8].

To begin to address these challenges, we will introduce a lightweight and flexible validation testing and framework called *SciUnit* in Sec. 2. Many of these challenges have to do with coordination between scientists. To begin to address this, we then describe a community workflow based on widely-adopted social coding tools (here, Github) and a lightweight community portal called *SciDash*, in Sec. 3. The overall goal of this work is to help scientists generate and examine tables like the one central 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. We discuss adoption strategies and directions for future research into scientific model validation practices in Sec. 4.

## 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.

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 [7], and is increasingly regarded as the *de facto* language of open source scientific computing. It is easy to read and has a simple object system that we use to specify the interface between tests and models (challenge 2, see below). It supports calling into other popular languages, including R, MATLAB, C and Java, often more cleanly than they support calling into Python (challenge 3). The IPython notebook is a powerful web-based "read-eval-visualize loop" that we will use to support interactive table visualization, and it also permits using other languages on a per-cell basis [6]. Together, this

```
1  class PositionTest(sciunit.Test):
2    """Tests a planetary position model based on
         positions observed on night n given the
         positions in the n-1 previous nights.
3    Observation: {
4      'histories':list[list[Position]],
5      'positions':list[Position]]
6    }"""
7    required_capabilities = [PredictsPlanetaryPosition]
8    def generate_prediction(self, model):
9      return [model.predict_next_pos(obs_history)
10       for obs_history
11       in self.observation['histories']]
12
13   def compute_score(self, observation, prediction):
14     obs_positions = observation['positions']
15     return pooled_p_val([abs_distance(obs, pred)
16       for (obs, pred)
17       in zip(obs_positions, prediction)])
```

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

makes writing wrappers around tests and models written in other languages relatively simple. We anticipate making it nearly automatic as future work.

A SciUnit validation test is an instance of a Python class inheriting from `sciunit.Test` (line 1). Every test takes one required constructor argument (inherited from the base class): the observation(s) against which the test will validate models. In our example, this is a dictionary, documented per Python conventions on lines 2-6. To create a *particular* position test, we instantiate this class with particular observations. For example, the subset of cosmologists interested specifically in Saturn might instantiate a test by randomly chunking observations made about Saturn's nightly position over time as follows:

```
h, p = randomly_chunk(obsvd_saturn_positions)
saturn_position_test = PositionTest(
  {'histories': h, 'positions': p})
```

The class `PositionTest` defines logic that is not specific to any particular planet, so it is contained in a repository shared by all cosmologists called `cosmounit`. The particular test above would be constructed in a separate repository focused specifically on Saturn called `saturnsuite`. Both would be collaboratively developed by these (overlapping) research communities in source code repositories on Github.

Classes that implement the `sciunit.Test` interface must specify two methods: a method for extracting a prediction from a model, and a method for producing a score from that prediction. Predictions are extracted by a separate method to make it easier for statisticians to write new tests for which only the goodness-of-fit metric differs, not the method by which the prediction is extracted (challenge 5).

The `generate_prediction` method (lines 8-11) is passed a *model* as input and must extract a *prediction*. A model is an instance of a class inheriting from `sciunit.Model` and a prediction can be any Python data structure. To specify the interface between the test and the model, the test class specifies a list of `required_capabilities` (line 7). A capability specifies the methods that a test may need to invoke on the model to extract a prediction, and is analogous to an *interface* in a language like Java. In Python, capabilities are written as classes with unimplemented members, shown in Figure 3. Classes defining capabilities are tagged as such by inheriting from `sciunit.Capability`. The test in Figure 2 repeatedly uses this capability on lines 9-11 to return a list of predicted positions for each observed history (using a list comprehension for brevity). A planet's position is represented using a standard celestial coordinate system specified

```
1  class PredictsPlanetaryPosition(sciunit.Capability):
2    def predict_next_pos(self, history):
3      """Takes a list of previous Positions and produces
           the next Position."""
4      raise NotImplementedError()
5
6  class LinearPlanetModel(sciunit.Model,
7      PredictsPlanetaryPosition):
8    def predict_next_pos(self, history):
9      return linear_prediction(history)
```

**Figure 3: An example capability and a model class that implements it in `cosmounit`.**

within `cosmounit` by the class `Position`.

A model class implements a capability by inheriting from it and implementing the required methods (Figure 3). The scope of a model class is identified by the capabilities it has. A particular model is an instance of such a class:

```
lin_saturn_model = LinearPlanetModel()
```

Once a prediction has been extracted from a model, the test class must compute a *score*. The framework invokes the `compute_score` method with the observation provided upon test instantiation and the prediction just generated. On lines 13-17 of Fig. 2, the test class constructs a list of distances between the observed and predicted positions, then determines a pooled $p$-value to determine the goodness-of-fit (the details are omitted for concision). A $p$-value is represented as an instance of `sciunit.PValue`, a subclass of `sciunit.Score` that has been included with SciUnit due to its wide use across science.

This illustrates a key difference between unit testing, which would simply produce a boolean result, and our conception of scientific validation testing (challenge 1). A score class must induce an ordering, so that a table like that shown in Figure 1 can be sorted along its columns, and it can specify a normalization scheme so the cells can be color-coded.

Te `judge` method of a test can be invoked to compute a score for a single model:

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

This method proceeds by first checking that the provided model implements all required capabilities before calling the `generate_prediction` method followed immediately by the `compute_score` method. A reference to the test, model, observation, prediction and other related data the test provides (none here) are available as attributes of the score.

To produce a comprehensive test suite, the contributors to `saturnsuite` would instantiate a number of other tests and then create an instance of the `TestSuite` class:

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

They would also instantiate a number of models. A test suite can be used to judge multiple models at once, if they satisfy the union of the capabilities required by the constituent tests (challenge 2). The result is a *score matrix*:

```
sm_matrix = saturn_motion_suite.judge([
  lin_saturn_model, ptolemy_model, kepler_model, ...])
```

If constructed inside an IPython notebook, a score matrix can then be visualized as an interactive table, much like the one shown in Figure 1. Scientists can sort by column and click on tests, models and scores to get more information on each, including documentation, authorship information, related papers and other related data extracted from the underlying classes (details omitted for concision).

# 3 COLLABORATIVE WORKFLOW

Social coding tools like Github allow each scientist to work on their own fork of a repository like `saturnunit` where they can examine the consequences of their own contributions on the tests and models that have previously been developed. For example, an experimentalist who has gathered new data can instantiate a new test or refine an existing one with more accurate observations. As long as the interface between the test and the model remains the same, the scores for prior models can be recomputed entirely automatically, requiring no coordination with model developers (challenge 4). When a modeler develops a new model, she can validate it against all the tests it is capable of taking. If a statistician develops a better goodness-of-fit metric, it too can existing capabilities, and so a comprehensive score table can be generated easily, as discussed above. If a scientist is not interested in certain tests, she can locally remove them to recalibrate her view of the relative merits of different models (challenge 5).

Once satisfied with their contributions to the repository, these scientists might submit a *pull request* to the central repository (either separately or together with the submission of a conventional paper). Anonymous reviewers tasked by the "editors" of the repository would then subject the pull request to appropriate peer review before committing it to the "academic record". A model that performs well across tests in a collaboratively developed suite could more directly and believably claim (e.g. to reviewers) to be a coherent, valid model of, for example, Saturn's motion.

To help scientists discover and coordinate control over these repositories, we are developing a collaborative portal called *SciDash* (*http:// scidash.org/*). It is essentially a wiki pointing scientists to the common and suite repositories designated as "primary" by early adopters in their research areas (who serve as *de facto* initial editors as well). SciDash also supports loading IPython notebooks containing score tables stored in these repositories into the browser directly, avoiding the need to clone the repository locally. To modify the suite (e.g. to add the model a scientist is developing to it), a scientist can simply fork the repository by clicking a button at the top of this view. We are exploring ways of loading forks directly into a cloud-based IPython notebook system, so that the complexities of Git and installation of Python and the packages required by the repository are left behind the scenes (challenge 4). SciDash discovers public forks of repositories that it is indexing and lists them to help scientists get a better sense of who is working on the problems they are interested in and what partial results are publicly available, fostering collaboration.

As with many pieces of cyberinfrastructure, early adopters may benefit less than late adopters due to network effects. To incentivize adoption, we look to an increasingly common scientific practice: modeling competitions. Many research communities use these to periodically determine the state of the art, resulting in a form of review paper. Funding sources often include small monetary incentives. SciDash could be used to run these kinds of competitions on a continuous basis atop less *ad hoc* infrastructure (and thus require less effort and central planning from organizers) and we are actively organizing such competitions.

# 4 DISCUSSION & FUTURE DIRECTIONS

Instead of attempting to use an existing general-purpose testing infrastructure, we have designed what is essentially a domain-specific language that captures the scientific model validation process in more familiar terms (challenge 6). SciUnit can be used by individual scientists to organize their workflows and also within a loosely collaborative workflow, mediated by a social coding tool like Github and an easy-to-use portal, SciDash, based on the IPython system [6].

While we discuss a simple example based on planetary movement here, we have applied this framework to more realistic problems in neurobiology in collaboration with a large-scale informatics project, OpenSourceBrain, whose developers are enthusiastic about developing a practical pipeline for precisely characterizing the scope and validity of the dozens of models they currently maintain (case study in preparation). The tools we describe could also be used for a variety of other scientific disciplines, including within software engineering to validate quantitative models of, for example, software cost or component reusability.

This paper identifies potential synergies between software development and model validation practices, and provides a proof of concept to argue that a test-driven development methodology could help improve scientific workflows. The modern scientific model validation process has not yet been extensively studied by the research community. Precisely characterizing how scientists today attempt to answer the questions in the introduction, and evaluating the effectiveness of tools like the ones we propose, are likely fruitful avenues for future empirical research.

The core SciUnit framework has been fully developed and is available at http://sciunit.scidash.org/. SciDash is under active development as of this writing.

# 5 ACKNOWLEDGEMENTS

We thank Sharon Crook, Shreejoy Tripathy and Padraig Gleeson for their many helpful discussions. This work was supported in part by NIMH grant R01MH081905. The content is solely the responsibility of the authors and does not necessarily represent the official views of the NIH.

# 6 REFERENCES

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

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

[3] J. Howison and J. Herbsleb. Scientific software production: incentives and collaboration. In *CSCW*, pages 513–522. ACM, 2011.

[4] A. E. Jinha. Article 50 million: an estimate of the number of scholarly articles in existence. *Learned Publishing*, 23(3):258–263, July 2010.

[5] C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Transactions on Software Engineering*, 35(4):534–550, 2009.

[6] F. Perez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[7] M. F. Sanner et al. Python: a programming language for software integration and development. *J Mol Graph Model*, 17(1):57–61, 1999.

[8] J. Segal. Models of scientific software development. In *SECSE*, May 2008.

[9] H. Siy and L. Votta. Does the modern code inspection have value? In *ICSM*, pages 281–289, 2001.