"OpenQuake: Calculate, share, explore"

Testing procedures adopted in the development of the risk component of the OpenQuake-engine

Authors
Anirudh Rao[1]

[1] GEM Model Facility
via Ferrata, 1
20133 Pavia
Italy

Email address (for all the authors):
<name.surname>@globalquakemodel.org

Citation
Please cite this document as:
Rao, Anirudh (2015). Testing procedures adopted in the development of the risk component of the OpenQuake-engine *Global Earthquake Model (GEM) Technical Report 2015-04.*
doi: 10.13117/GEM.OPENQUAKE.TR2015.03, 27 pages.

Third printing, March 2015

Contents

Part I

Introduction

1. Software Testing

The current document describes the testing procedures adopted in the development of the hazard component of the OpenQuake-engine (OQ-engine), the open source hazard and risk software developed by the Global Earthquake Model initiative.

Nowadays seismic hazard analysis serves different needs coming from a variety of users and applications. These may encompass engineering design, assessment of earthquake risk to portfolios of assets within the insurance and reinsurance sectors, engineering seismological research, and effective mitigation via public policy in the form of urban zoning and building design code formulation.

Decisions based on seismic risk results may have impacts on population, properties and capitals, possibly with important repercussions on our day-to-day life. For these reasons, it is recommendable that the generation of hazard models and their calculation is based on well-recognized, state-of-the-art and tested techniques, requirements that must be reconciled with the need to regularly incorporate recent advances given the progress carried out within the scientific community. The features described below contribute to fulfill these requirements:

- Software should have a modular and flexible structure capable of incorporating new features and - as a consequence - offer users the most recent and advanced techniques. In very general terms, modularity is the level to which a component of a system can be moved, replaced or reused. In software design, modularity means the separation of the software into smaller independent components that can be implemented, maintained and tested easily and efficiently.
- Software should have and extensive test coverage which captures possible errors and avoids regressions (i.e. unexpected behaviors introduced by new features). Software testing (Myers et al., 2012) is an important, complex and vast discipline which helps in developing methods and processes aimed at certifying the extent to which a computer code behaves according to the original design intent and user specifications.

The OQ-engine includes different levels of modularity. The first is the one separating the engine itself into a number of libraries (see Figure 1.1), each one containing well identified knowledge, objects and methods (e.g. the OQ-hazardlib includes objects and methods needed to compute probabilistic seismic hazard and the OQ-risklib contains methods to compute scenario and probabilistic seismic risk). The second one pertains to the data model adopted in the development of each library as a result of the abstraction process. According to Berkes (2012) scientific

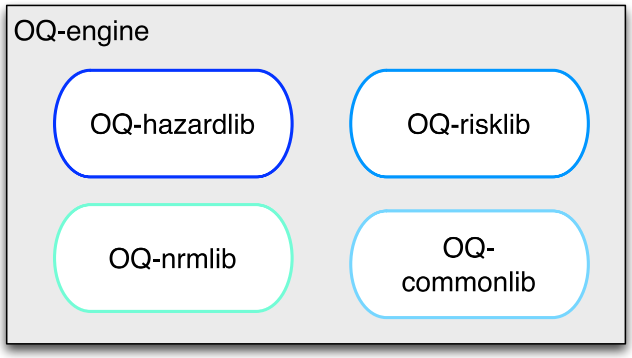

Figure 1.1 – *A schematic describing the main components of the OpenQuake-engine software.*

software must be:

- Error proof
- Flexible and able to accommodate different methods
- Reproducible and re-usable

1.1 Testing and Quality Assurance

Despite the distinction between software testing (in some cases also called Quality Control) and Software Quality Assurance (SQA) being somewhat vague and partly open to personal judgment, it's clear that SQA is a more comprehensive and overarching process than software testing. SQA aims at the definition of the best processes that should be used to provide guarantees that user expectations will be met. Software testing focuses instead on detecting software faults by inspecting and testing the product at different stages of development.

1.1.1 Testing

Software testing can be implemented at different stages of the development process, with varying strategies to approach the problem. The OQ-engine and the associated libraries are developed following an agile paradigm. This development strategy is organized in a way that the creation of the real code is completed in parallel and fully integrated with the software testing process.

The software engineering community provides a wide range of testing levels and typologies. In the current document we consider just a portion of them with the specific intent to illustrate the standards used in the development of the OQ-engine and particularly of its hazard component.

1.1.2 Quality Assurance

From the IEEE "Standard for Software Quality Assurance Processes": *Software quality assurance is a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes. A key attribute of SQA is the objectivity of the SQA function with respect to the project. The SQA function may also be organizationally independent of the project; that is, free from technical, managerial, and financial pressures from the project.* In this document we are not covering topics related to SQA since this would go beyond its scopes.

1.2 Document structure

The document is organized into four main chapters and two appendixes.

In the current chapter we provide a very brief and general introduction to software testing with a focus on the testing of scientific software.

In the second chapter we describe the module, or unit, testing, the acceptance tests adopted in the development of the OQ-engine and we discuss some examples.

In the third and fourth chapters we illustrate tests comparing the results computed with the OQ-engine against the ones computed using different probabilistic seismic hazard analysis software.

Appendix **??** provides details on the PEER tests implemented in the OpenQuake hazard library (OQ-hazardlib).

Part II

Unit Tests

2. Unit Testing in the OpenQuake-engine

This chapter provides an introduction to the module (unit) testing procedures (Myers et al., 2012) and describes estensively examples of tests implemented in the OQ-engine.

2.1 Unit-Testing: An Overview

At the first level of the code testing process is the practice of "unit-testing". This process is a central tenet of test-driven software development and is widely established as a means of "best-practice". Before looking closely at the OpenQuake-engine approach to unit-testing it is important to establish what are the precise objectives of the unit-testing process and the benefits (and limitations) that it brings.

2.1.0.1 Correctness of Implementation

This objective is obviously the primary goal of unit-testing, to ensure that each function of the code is operating in the manner expected by the developer. "Correctness", in this case, requires that the function produces both the correct output, but also if there are cases in which function may fail then the means of failure should be predictable. The following is a relatively simple example of how a unit-test relates to a function:

Consider a simple function to multiply two numbers and take the logarithm of the result. A relevant analogy may be that of a magnitude scaling relation calculation, in which both a rupture length and rupture width are required, and the logarithm of the area may be needed by the function itself. In this circumstance a negative value in either of the two inputs would result in a calculation error. This could be coded in the following manner.

```
def get_log_area(length, width):
    if (length < 0) or (width < 0):
        raise valueError("Both inputs must be positive")
    else:
        return log10(length * width)
```

From the description above it is evident that the user requirements inform the manner in which the function should behave (i.e. negative values cannot be tolerated). To ensure that the function is operating correctly, we wish to write a set of tests that will confirm the behaviour is correct:

1. If both a and b are equal to 10.0, then the function should return 2.0
2. If $a = -1$ and $b = 10$ the function should raise an error reporting the stated message "Both inputs must be positive".
3. If $a = 10$ and $b = -1$ the function should raise an error reporting the stated message "Both inputs must be positive".
4. If $a = -1$ and $b = -1$ the function should raise an error reporting the stated message "Both inputs must be positive".

A unit-test for this function is an additional function that will check that both cases are satisfied, and will report an error if not.

A comprehensive unit-test suite for a software may fulfil two objectives: **line coverage** and **parameter coverage**. The former should ensure that, in as far as possible, every line (or statement) in the code is executed at some point in the testing process. The latter should ensure that the behaviour of the function is predictable when supplied with "unusual" parameters. In the above example, both objectives are satisfied by the tests. The first test will result in a positive valued "area", thus executing the second branch of the logical path, the second test will result in a negative area and will execute the first logical branch. Therefore all lines of the code are covered and the line coverage is complete. We also see that in this simple example there are four possible cases: i) a is positive and b is positive, ii) a is positive and b is negative, ii) a is negative and b is positive, and iv) both a and b are negative. Only the first case is valid, therefore the first test ensures that they provide the correct answer (usually verified by independent means), whilst the remaining tests should ensure that the function raises the correct error. Thus the full parameter space of the input is ensured.

The above case is, of course, trivial; however, as shall be seen in due course, this same process can be applied in more complex contexts. Furthermore, the same unit-testing approach can be applied not only to individual components within the PSHA calculation, but also to full calculations, essentially verifying that the hazard curve produced by the full PSHA calculator is in agreement with that produced independently (sometimes by hand).

2.1.0.2 Identifies Problems Prior to Software Release

This advantage is largely self-explanatory, but for many software projects this can reduce the possibility of requiring *a posteriori* fixes to the code (patches). By compiling a comprehensive suite of unit-tests, and following a software development and release process that should automatically run the tests at the point of packaging, this should ensure that new features added to the software cannot inadvertently break other components.

2.1.0.3 Facilities Improvements in Performance

In the creation of software intended to perform demanding scientific calculations, like those commonly associated with PSHA, the issue of computational performance and efficiency is a major one. There is a continuing need to improve the speed and reduce the work required to undertake the PSHA calculation. To implement improvements it is necessary to ensure that optimisations do not modify the outputs of the calculation, only the speed at which they are performed. The unit-testing is absolutely fundamental to this process as optimisation cannot be undertaken readily without a means to ensure the calculation outputs have not changed. This point was a critical motivation behind the transition from the OpenSHA basis of the OpenQuake hazard calculation engine prior to version 1.0, to the current OpenQuake hazard library.

2.2 Continuous Integration

OQ-engine is developed and packaged within a "continuous integration" system (https://ci.openquake.org/), based on the open-source software "Jenkins" (http://jenkins-ci.org/). Continuous integration is used in large software projects to run a full test suite of the complete software, either at fixed time intervals or, as in the current case, when any new code is committed to the repository. The continuous integration system will do the following:

1. Run the full set of unit-tests for all code in all of the linked repositories. This will include the main (or "master") branch of the software repository, i.e. the one that will be used for packaging of the software, as well as some development branches.
2. Run a test of the software installation. This test will install the software on a dedicated platform and check that the installation of the software is successful. This test also ensures that if changes occur in the dependency packages, and these changes affect or compromise the installation and operation of the software, these problems are recognised immediately.
3. The software will also run standard Python tests for quality of code, compilation of documentation etc.
4. Several long-running tests may also be run. These implement larger scale seismic hazard and risk calculations designed to test the overall performance of the engine.

If at any point the tests should fail, the OpenQuake development team will be notified automatically. This ensures that software that is failing any of the tests will remain on the main branch of the repository for the minimum amount of time possible. Furthermore, if the continuous integration tests fail, the new code will not be integrated into the nightly package of the software.

2.3 The OpenQuake Hazard Library: Unit Tests

The unit-test suite for the OpenQuake-engine hazard library consists of three types of tests: i) simple tests for individual functions to verify the correctness of implementation ("component testing"), ii) simple tests of the full calculators for PSHA ("method testing"), and iii) "acceptance" tests, which provide a basic quality assurance check for the each of the three main calculators.

2.3.1 Component Testing

The unit-testing at the component level breaks the functions into simple calculations whose results can be verified by hand. These tests, similar in nature to that illustrated previously, provide the majority of the line and parameter coverage needed to ensure a robust code. To illustrate the comprehensive nature of the coverage we consider the example of the functions to undertake calculations of geodetic distance between two points[1].

Whilst not necessarily a complex function in itself, the distance between two points on the Earth's surface is a critical component of the software that is frequently called at several points of the PSHA process. Therefore, it is critical that the function operates correctly and its behaviour under extreme cases is understood. Thus, this relatively simple function is verified in the following cases:

- `test_LAX_to_JFK`
  Checks that a correct geodetic distance is calculated for two known locations on the Earth. This value is verified against an implementation of the algorithm provided by an online geodetic calculation tool.
- `test_on_equator`
  Checks that the correct distance is provided for two points located on the equator.
- `test_along_meridian`
  Checks that the correct distance is returned for two points located along the same meridian.
- `test_one_point_on_pole`
  Verifies the distance calculations for two points assuming one point is located at the geographic pole.

[1] This function can be found here: https://github.com/gem/oq-hazardlib/blob/master/openquake/hazardlib/geo/geodetic.py

- `test_small_distance`
  Verifies that two points separated by a distance within the floating point error are considered to be separated by zero km
- `test_opposite_points`
  Verifies the correct distance between two points in different longitudinal hemispheres (i.e. checks that the distance crosses the international dateline correctly).
- `test_array`
  Verifies the correct distances between two set of points
- `test_one_to_many`
  Verifies the correct distances between one point and a set of points.

The test suite for this one function is illustrative of several key components of the unit-testing. First is the use of an independent tool to provide the expected values of the calculation under simple conditions. Second is the use of "extreme cases" such as polar locations, or across the International Dateline. These ensure that the function can be global in application.

The nature of the interdependencies between the functions also means that one a functions own unit-test is verified, the function can then form the basis for testing other conditions. So for example, the geodetic distance tools also contain a method to calculate the minimum distance between a collection of points and a single point. Rather than requiring new expected distances for the different conditions, the geodetic distance function can then be used to construct tests for functions that utilise it. This makes the testing process more efficient, and reduces the need to write large numbers of tests in order to ensure correct behaviour of the function.

2.3.2 Ground Motion Prediction Equation (GMPE) Testing

The implementation process for ground motion prediction equations requires careful consideration, as it is in this area that new features may be expected to be added regularly, and where contributions from third parties are more likely to be incorporated into the software. Furthermore, in most cases the expected values of the functions may only be obtained from independent implementations of the GMPE. These expected values take the form of test tables, which are simple comma-separated value (csv) files that provide the expected values and standard deviations of the ground motion prediction equation for an exhaustive combination of parameters for the predictor variables. These should be sufficient to ensure that every part of the GMPE is covered within the test. An example test table is shown in Figure 2.1.

To ensure the most objective testing strategy, we aim for the test tables to match the GMPE creator's own implementation of the GMPE, in as far as possible. Therefore we prefer to solicit input from the authors of the GMPE. This will often take one of two forms. We ask that the authors can provide test tables, in a convenient format, or that they provide their own software implementation of the GMPE, from which we will then generate the test tables. Input from the GMPE authors is highly desirable within this process as it can help resolve issues that are perhaps ambiguous within the original publications of the GMPE and it can identify errors and bugs in the author's own implementation. The full workflow for GMPE implementation in OpenQuake-engine is shown in Figure 2.2.

The GMPE unit-tests themselves are designed to be simple for the user to create once the test tables are provided. Ideally the expected values should match the implementation values to within the test precision (typically permitting a difference of 10^{-7}). In some cases, however, it may not be possible to match the desired level of precision and therefore the tests permit the maximum discrepancy level (as a percentage) to be specified. Discrepancies may arise due to rounding of the coefficients within published tables, but ideally the tolerable discrepancy between an expected and predicted value should be not more than one tenth of one percent.

As is shown from Figure 2.2, once the GMPE test tables are created, the GMPE implementa-

./qareport/pictures/test_tables_screen_capture.pdf

Figure 2.1 – *Example GMPE test table used by OpenQuake*

./qareport/pictures/gmpe_implementation_flowchart.pdf

Figure 2.2 – *OpenQuake-engine GMPE Implementation Process*

tion should then be checked against the unit-tests. If discrepancies cannot obviously be resolved the author may then be contacted for clarification. Once the unit-tests pass the code is then submitted for review by (typically) one or more members of the software development team and one or more of the scientific team. This may help identify issues such as inefficiencies or unclear code. Once the submission is accepted by both the scientific and IT reviewer the code is merged into the main repository. This will then trigger a full test from the continuous integration system described previously.

2.4 Summary

In this section we have outlined both the process and the key benefits of developing comprehensive unit-tests for OpenQuake-engine, as well as outlining the operation of the continuous integration system, which should ensure that code with the potential to break the tests cannot be packaged and released. The unit-tests themselves have not been discussed in detail as nearly one thousand tests are executed during the unit-test process. However, to view the comprehensive set of tests, the reader is encouraged to refer to the full test-suite, which is open and available on the OpenQuake code repository (https://github.com/gem/oq-hazardlib/tree/master/openquake/hazardlib/tests). Furthermore, we have also discussed how the OpenQuake-engine development tries to facilitate correct implementation of features such as ground motion prediction equations. For relatively simple conditions, a selection of PEER tests (Thomas et al., 2010) are built into the testing process, making OpenQuake-engine unique amongst other hazard software in integrating the verification into the development process. The following chapters will expand in greater detail upon the additional hazard curve benchmark tests, which both follow and expand upon the PEER testing process.

Part III

Acceptance Tests

3. Framework for Acceptance Testing

4. Acceptance Tests for the Scenario Risk Co

5. Acceptance Tests for the Classical Risk Co

6. Acceptance Tests for the Event-Based Risk

Part IV

Benchmark Tests

7. Comparing Loss Curves from the Classical

8. Comparison with Other Softwares

Part V

Performance Tests

9. Demos

Books
Articles
Reports

10. Stress Tests

Bibliography

Books

Myers, G. J., C. Sandler, and T. Badgett (2012). *The art of software testing*. Wiley and Sons, Inc. (cited on pages 7, 13).

Articles

Other Sources

Berkes, P. (2012). *Writing robust scientific code with testing (and Python)*. Euroscipy. URL: http://archive.euroscipy.org/talk/6634 (cited on page 7).

Thomas, P., I. Wong, and N. A. Abrahamson (2010). *Verification of Probabilistic Seismic Hazard Analysis Computer Programs*. PEER Report 2010/106. College of Engineering, University of California, Berkeley: Pacific Earthquake Engineering Centre (cited on page 19).