Journal of Seismology, Accepted, `http://link.springer.com/article/10.1007/s10950-018-9731-3`
The final publication is available at link.springer.com

Seismology Software: State of the Practice

W. Spencer Smith · Zheng Zeng · Jacques Carette

Available on-line: February, 2018

Abstract We analyzed the state of practice for software development in the seismology domain by comparing 30 software packages on four aspects: product, implementation, design and process. We found room for improvement in most seismology software packages. The principal areas of concern include a lack of adequate requirements and design specification documents, a lack of test data to assess reliability, a lack of examples to get new users started, and a lack of technological tools to assist with managing the development process. To assist going forward, we provide recommendations for a document driven development process that includes a problem statement, development plan, requirements specification, verification and validation (V&V) plan, design specification, code, V&V report and a user manual. We also provide advice on tool use, including issue tracking, version control, code documentation and testing tools.

Keywords Seismology, Scientific Computing, Software Quality Model, Analytic Hierarchy Process (AHP), Software Engineering

1 Introduction

Seismology is the scientific study of earthquakes and seismic waves through and around the earth. Many valuable software tools have been created by end user developers in the seismology community. The developed software tools automate the task of processing large amount of data, analyzing patterns with complex calculations, simulating seismic waves, making seismograms, etc. As in other scientific domains, the software provides crucial support to the important work of seismologists. Given the importance of the software, and its value to various scientific and engineering communities, it is worthwhile to discuss whether there is room for improvement in the quality of existing seismology software.

Computing and Software Department
McMaster University, Hamilton, Ontario L8S 4L7, Canada
E-mail: smiths@mcmaster.ca

In recent years, software researchers and scientists have found that many Scientific Computing (SC) software tools show room for improvement in terms of quality (Wilson et al 2013). This is, in part, explained because many SC tools are developed by scientists or researchers (Carver et al 2007; Kelly 2007; Sanders and Kelly 2008; Segal 2007a,b; Segal and Morris 2008), rather than software engineers or computer scientists. These scientist developers are known as “professional end-user developers” (Segal 2007b). They work in their own domain, as an expert, developing tools to help with their professional goals. Given the typical background of the end user developer, software engineering methods are not widely applied in scientific communities (Kelly 2007, 2013; Segal and Morris 2008). Moreover, some of the developers tend to be averse to “process-oriented” development processes (Carver et al 2007).

Our goal is to better understand, from a software engineering perspective, the current state of practice for seismology software. We build on our previous work measuring qualities in other scientific domains, including Geographic Information Systems (Lazzarato et al 2015), mesh generation (Smith et al 2016b, 2015a), psychometrics (Smith et al 2015b) and oceanography (Smith et al 2015c). The authors of this paper are not domain experts in seismology, but they have been working in the software industry, or doing software engineering research, for a combined total of more than 62 years. We develop a systematic methodology to assess quality, but we exclude comparisons based on the functionality provided by the available software packages. Given that we are outsiders to the seismology community, we can be objective. We aim to determine why some software packages achieve good quality while others do not. We will also provide suggestions for improving seismology software quality in the future. The goal is to provide insight to the communities on what is working well and what area could be improved going forward.

In Section 2, we will present background information for this study. Our methods will be provided in Section 3. Project results will be shared in Section 4. In Section 5, we provide our recommendations, including an overview of a document driven design process. Conclusions are found in Section 6.

2 Background

In this section, we summarize our software quality definitions and we introduce our use of the Analytic Hierarchy Process (AHP). AHP provides us with a systematic means to compare software packages across different quality attributes.

2.1 Software Quality Definitions

Software is very different from traditional manufactured products, which can be touched and unambiguously measured. For software, we need to explicitly state the definitions for the qualities of interest. We adopt the software quality definitions from Ghezzi et al (2003). The qualities are presented below in the same order as in Ghezzi et al (2003), bookended by two additional qualities of interest for this project: installability and reproducibility).

Installability Before a user can take advantage of a software product, installation has to occur. Installability is the degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment (ISO/IEC 2010). Installability can be achieved by giving installation instructions, or an automatic installation tool or script, or executable binary file, etc. An easy and verifiable installation is preferred by users. Achieving good installability is important for professional performance (Simmons and Sea 1994).

Correctness The term correctness is often mentioned as a degree to which software meets the requirement specification (IEEE 1990). In this definition, the specification has to exist to determine whether the software is compliant with it or not. The functional requirement specifications are rigidly required by projects in software companies; but, in SC software development, this is not always the case (Sanders and Kelly 2008; Segal 2007b). Strictly speaking, especially in some algebraic specification languages (IEEE 1990; Bidoit et al 2002), correctness is a mathematical property that represents the equivalence between a program and its specification.

Verifiability This quality is sometimes referred as testability, since the focus is on measuring how easily the properties of a software can be checked or proven. Theoretically, program verification is part of the broader area of formal methods (Almeida 2011), which along with correctness can be established by mathematical models in rigorous software development. In practice, we often consider approaches that are more feasible, such as testing, to build confidence, rather than formal analysis methods.

Reliability As we know for complex software there is no such thing as “bug-free” software, so reliability is defined to be a measurement of the extent to which the user can depend on the software, even if it is not absolutely correct for all possible inputs. Formally, reliability is a measure of the probability that the software performs its required function under stated conditions for a specified period of time (IEEE 1990).

Robustness When a software product is put into use, it does not always get the expected command or data. Robustness then is about the ability of the software to deal with unexpected input. It is not easy to have a complete definition, but we can expect it to have “reasonable” behaviour in the presence of invalid inputs or an unexpected environment. This quality is related to error tolerance.

Performance Performance is typically about memory usage, storage, and speed. Usually performance is considered as an important non-functional requirement, which is expected to be optimized after functional requirements have been met.

Usability This terminology is also known as User Friendliness (Ghezzi et al 2003). This quality depends on the user interface design, regardless of whether the interface has a GUI or not. Helpful support, such as a user manual or a user community, also contributes to a software’s usability. Formally, usability reflects the extent of ease with which users can learn to operate and interpret the output of the software (IEEE 1990).

Maintainability In modern software engineering, maintainability has become one of the most important aspects for software. The maintenance period is the longest working phase and the most costly, being estimated to constitute at least 50% of the total project lifecycle costs (van Vliet 2000, p. 450). Maintainability is a crucial factor to software's success. Maintainability is determined by documentation artifacts, management processes, tools, development standards, etc. Maintainability can typically be considered from three perspectives: corrective, adaptive and perfective (Ghezzi et al 2003). It is a complex measurement to assess the health of the development process and sustainability of the product delivery. Maintainability is the ease with which a software product can be modified for correction or improvement (IEEE 1990).

Reusability In terms of software products, rather than the development process, reusability often refers to reusable components. Generally speaking, reusability is about the degree to which a software product or a component, can be used in another software system (IEEE 1990).

Portability Portability is the ease with which software, or a component, can be transferred from one platform (or environment) to another. This ability can be achieved by either adopting a platform independent programming languages or branched development for different environments. Portability is often required by those software products designed as tools or invokable as libraries.

Understandability (of code) Understandability is considered as an "internal product quality" (Ghezzi et al 2003), since the user does not directly see this quality. Understandability measures the ease with which a new developer can understand the design and source code. Good understandability contributes to maintainability, and provides critical information for verifiability.

Interoperability Interoperability plays an important role in highly integrated software, where the workflow involves multiple applications. There are different forms of interoperability, such as the ESB (Enterprise Service Bus) integrated system, which uses a standardized message format; or the Microsoft Office products, which have internal connections with each other. Here, we mean syntactic interoperability, which is the ability of more than one software packages to exchange their information and use it.

Visibility/Transparency This terminology in software engineering is about the development process. A well formed development process will clearly define its working phase to coordinate the development team. Usually it means that, under a certain development process, the current progress and status are visible to the whole team. Visibility is intended to ensure that everyone knows where they are standing and where they are going. This kind of transparency reduces the possibility of project failure due to some fatal mistakes made by a single team member.

Reproducibility The aim of reproducibility is to provide enough information to enable anyone to verify scientific research results. Davison (2012) observes that reproducibility is part of the definition of the scientific method. Reproducibility is often archived by manual recording of details on the development environment (mainly including run-time platform and test data), or automated tools such as the Madagascar framework (Fomel et al 2013) and Sumatra (Davison 2013, 2012).

2.2 Analytic Hierarchy Process

The Analytic Hierarchy Process (AHP) is a structured analysis technique developed for multi-criteria decision making by Saaty (Triantaphyllou 1995). AHP is used in a wide variety of decision making situations and for tasks such as prioritization, resource allocation, benchmarking and quality management (Forman and Gass 2001). We adopt AHP in our project because it is a proven systematic method that helps eliminate bias caused by personal perceptions and judgments (Saaty 2008-09-01). AHP has the benefit that it does not require that qualities like maintainability and usability be measured on the same scale. Instead, the measurement is the relative importance of one quality versus another, or the relative capability of one software option over another, for a given quality. The emphasis on pairwise comparisons means that the determination of each of the inputs to the AHP process does not have to consider the complexity of the overall problem, but rather can focus on the much simpler problem of comparing two things at a time.

The AHP process is composed of three steps (Saaty 1990): i) breakdown the complex problem (or goal) into a hierarchical structure of sub-problems (or sub-goals); ii) evaluate the elements by pairwise comparison; iii) calculate numerical properties of each alternative.

AHP analysis centres around n alternatives (the software packages) and m criteria (the software qualities). For a given project, pairwise comparisons are made between the m criteria to determine the priority ranking between the criteria. Similarly, pairwise comparisons are made between the alternatives to rank them for each criterion (quality). The pairwise comparisons are quantified using the 9 point scale proposed by Saaty

(Triantaphyllou 1995). Assuming a priority ranking between qualities, we can use AHP to produce a ranking for a set of software packages.

3 Methods

Although we have definitions of software qualities, we still need to determine how the qualities are to be measured. Practitioners and researchers have been trying to assess software quality through quality models dating back to the 1970s. Since then, quality models have become a well-accepted means to prescribe and describe software quality. Realizing that different intentions should have different quality models, we produced a hierarchical quality model for SC software, which combines both quantitative and qualitative approaches.

3.1 Overview

We selected the seismology domain for analysis since it involves heavy usage of SC software and there is a strong research community. In our analysis, we considered 30 seismology software packages. Most of them are listed by Incorporated Research Institutions for Seismology (IRIS) SeisCode (Lang 2013), which is a community repository for software used in seismology and related fields. Others were found from university research projects and some seismology research organizations, such as Computational Infrastructure for Geodynamics (CIG) (for Geodynamics 2014). The full list of the selected software, along with their url's, can be found in Appendix B.

Some of the software packages on the lists provide data manipulation assistance, such as data collection and data exchange. Given our interest in SC software, we did not select data manipulation programs; we selected software packages that are more computationally intensive, including digital seismic signal processing, data transformation, complex calculation and displaying. In the SeisCode community, the software we selected is categorized as data processing. Once the software packages were selected, we worked on our quality assessment step by step.

For the first step, we determined the template for measuring each software package against each of the quality criteria. In Section 3.2, we will see these measures should be objective and reliable. This template is the same as that used in the analysis of other SC software domains (Smith et al 2016b; Lazzarato et al 2015; Smith et al 2015a,b,c). The 56 questions that make up the grading sheet are provided in Appendix A and at `http://dx.doi.org/10.17632/67ncspgz8n.1`.

After we finalized our list of measurements, the second step was to download the selected software packages and setup virtual machines to create clean environments for their installations. Virtual machines were used for installation and testing because this simplifies adding and removing software. Each installation starts from a "clean slate;" we do not have to worry about strange and unexpected interactions with existing libraries. This also gives us a pure and fair test of installability, since an installation should work on a fresh operating system. After installation, each software package was then graded out of 10 on each quality criteria, following the grading template. All of the measurements are only based on information that can be found by searching on-line, as opposed to directly asking question of the developers. This approach provides a true measure of what a new user faces, and it creates a fair environment where all software is judged on equal footing.

In the third step, we used the grading results to construct our AHP tables. These tables compare each software package for each quality. Details on how the objective measures for each software package are converted into pairwise scores is found in (Smith et al 2015c). Given that we have no information about the priority of the qualities, since this differs between projects, we give each criteria the same weight. The AHP results can then be summed to give an overall quality ranking for the seismology software.

Finally, to verify the reproducibility of our methods, we did a peer review of 5 random software packages. A separate reviewer followed the same method and metrics as originally used. The two reviewers differed in the intermediate grading sheet results. However, the AHP method smooths out this problem because it turns the grades out of 10 into relative comparison. As long as the reviewers are themselves internally consistent, the relative differences between the two reviewers should follow the same pattern. In our verification exercise, the final AHP results from the different reviewers were consistent, which supports our belief that our results are reproducible and our method is reliable.

The initial grading of the seismology software took place in the Spring of 2014. The data for those packages that had changed was updated in the Summer of 2017.

3.2 Hierarchical Quality Model

Our quality model is based on the quality definitions provided in Section 2.1. For each quality we need to determine an associated set of *feasible* metrics. Our metrics were selected to satisfy the "seven criteria for a good metric" suggested by Watts (Gillies 2011):

1. Objective: The results should be free from subjective influences, no matter who the measurer is.
2. Reliability: The results should be precise and repeatable.
3. Validity: The metric must measure the correct characteristics.
4. Standardization: The metric must be unambiguous and allow for comparison.
5. Comparability: The metric must be comparable with other measures of the same criterion.
6. Economy: The simpler and, thus less expensive, the measure is to use, the better.
7. Usefulness: The measure must address a need, not simply measure a property for its own sake.

One of the consequences of applying the above criteria, is that some of the qualities can only be measured on the surface. The economic reality is that we only had between 2 to 4 hours to spend measuring each of the 30 software products. When a quality can only be surface checked, we make explicit note of this. Our measures also need to be feasible. We do not have access to information that the developers have not made publicly available. If there are details of the development process that are not publicly posted, they are not considered. By staying with what is publicly available, we achieve objectivity and standardization.

With our goal of assessing quality, we consider four aspects of seismology software quality, as shown in Table 1. The connection between the aspects and qualities is also provided in Table 1. Our quality model has 4 aspects with 13 criteria and 56 metrics in total.

3.3 Example

Here we take *installability* as an example, to show how a quality metric is quantified and measured. For many SC software packages, installability is often overlooked. As we have learned, these software packages are often developed by "professional end-user developer," who may not have fully tested their application in different environments. New users may encounter difficulties due to different system configuration. To measure installability for each software product, we installed it on a virtual machine, recorded the steps and answered the questions for installability given in Appendix A.

We give each metric a grade out of 10. In the current example, we have 7 questions for installability. Each of these questions will get about 1.5 points out of 10. The absolute answer of "Yes" or "No" will directly get points. For those questions that do not have an absolute "correct" answer, such as the count of the number of steps, we base the result on a comparison to the average response on this metric. Considering all the responses leads to an overall impression (an integer between 0 and 10). For example, by installing rfsyn, JRG and JEvalResp, we will have the grading results shown in Table 2. In this example, JEvalResp scored the highest because it was the only package that automated the installation and provided a means to validate it.

This process will continue for all 30 seismology software packages. After that, we convert the grading sheet to a 30×30 pairwise comparison table for each quality criteria by doing a relative comparison between each software package on each quality metric. In total, there are 13 pairwise comparison tables for AHP. The full grading results can be found in Appendix B. In the next section, we will share our AHP results for each quality aspect and measurement.

4 Results

Figure 1 shows a summary of the AHP results for all 4 aspects (product, implementation, design and process). The packages are sorted in ascending order, with a higher rating meaning better performance. In the subsequent sections, which go through each aspect in turn, we will use the same ordering as given in Figure 1.

Figure 1 highlights two software products (*PITSA* and *OregonDSP*) that we could not install, despite a significant amount of time attempting the installation. If they could have been installed, then these two products may have scored higher on the qualities of correctness, reliability, robustness and usability, since these qualities each include at least one measure that requires running the software. For instance, for reliability, one cannot judge whether the software breaks during initial testing, if one cannot run the software. The complete list of measures that require a successful installation are indicated in Appendix A. Even though not all qualities are accurately assessed when we cannot install the software, the decision was made to include the uninstallable software in the study for the following reasons: i) As Appendix A shows, only 7 of 56 questions used to measure the software actually require running the software; ii) For some of the measures that require running the software, we can do an approximate measurement, such as using user manual screenshots to judge the usability metric associated with "look and feel;" and, iii) If we think about the purpose of our AHP measurements to be aiding a user in selecting a software product, potentially reducing a few quality measures is not significant, since the user will never select a software package that they cannot install. In each case where the lack of installability has a potential impact on the measurements in question, this will be explicitly noted.

Table 1 Aspects of Quality

Aspect	This Aspect is About	Criteria (Qualities)
Product	The software package itself	Installability, Usability, Reliability
Implementation	The development or coding phase	Correctness/Verifiability, Maintainability, Understandability
Design	The design of the finished product	Robustness, Performance, Reusability, Portability, Interoperability
Process	The software development process	Visibility, Reproducibility

Table 2 Example of Installability Grade Sheet

Question	rfsyn	JRG	JEvalResp
Are there installation instructions?	Yes	Yes	Yes
Are the installation instructions linear?	Yes	Yes	Yes
Is there something in place to automate the installation?	No	No	Yes
Is there a means given to validate the installation?	No	Yes	Yes
How many steps were involved in the installation?	2	1	1
How many software packages need to be installed?	0	1	3
Run uninstall, if available. Any obvious problems?	No	No	No
Overall Impression (/10)	5	7	10

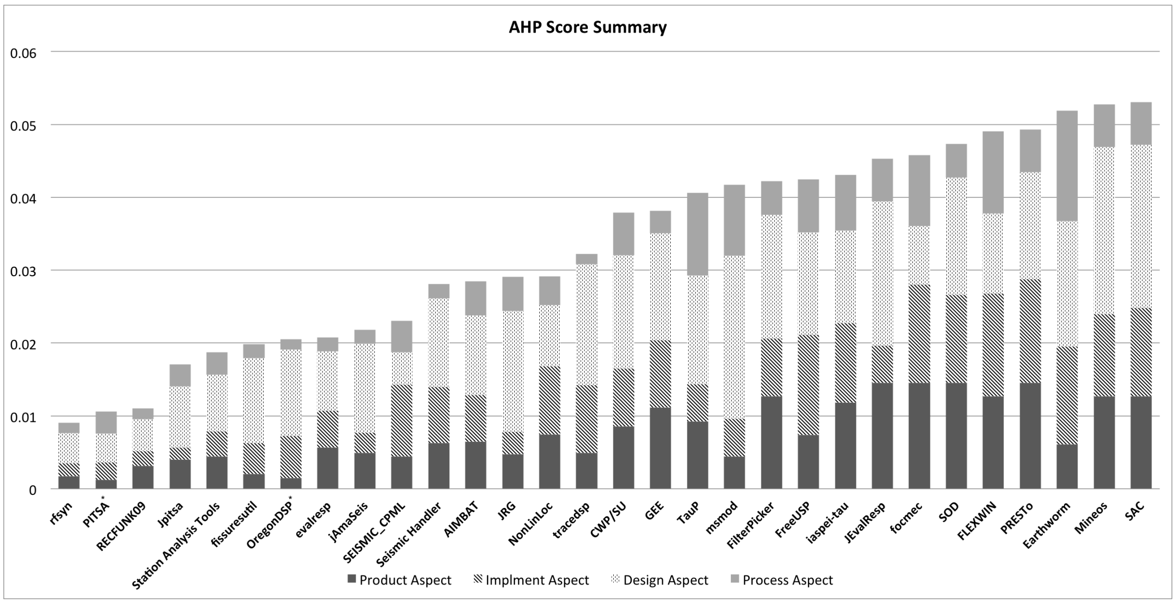

Fig. 1 Overall Quality Ranking (software marked with an asterisk (*) could not be installed)

Tables 3 and 4 summarize software packages that are alive and dead, respectively. About one third (11 out of 30) of the software is considered alive, since they were updated during the past 18 months; the rest (19 of 30) appear to be dead, or at least dormant. Most (22 out of 30) of the software is public, as defined by Gewaltig and Cannon (2012). Five of them are private software (Gewaltig and Cannon 2012) and the others are POC (Proof Of Concept) (Gewaltig and Cannon 2012) programs. (Gewaltig and Cannon (2012) proposed a revised 4-category classification system (Gewaltig and Cannon 2014), but for our purposes the simpler 3-part classification system meets our needs.) There are many (22 out of 30) command line tools; 5 out of 30 have a GUI (Graphic User Interface) and the rests are used as program libraries.

The software tools summarized here are all written by scientists or related researchers (including students during their Master's or PhD studies). This finding is consistent with many literature reports (Segal 2007a). About half of the software packages (16 out of 30) have related publications. There is no commercial software in our set of programs; most of them (28 out of 30) have source code available. We found C/C++ is the dominant (13 out of 30) programming language used for seismology software, followed by Java (9 out of 30). The other

Table 3 Alive Seismology Software Packages

Name	Type	UI	Language	Released	Updated
fissuresutil	Private	lib	Java	2012 Jun	2016 Jul
jAmaSeis	Public	GUI	Java	?	2017 Jan
SEISMIC_CPML	Public	cmd	FORTRAN	?	2017 Apr
AIMBAT	POC	cmd	Python	2012 Sep	2016 Aug
JRG	Private	GUI	Java	2003 Jul	2016 Apr
CWP/SU	Public	cmd	C	1987	2017 Apr
GEE	Public	GUI	Java	2003 Oct	2016 Nov
TauP	Public	GUI/cmd/lib	Java	?	2017 Mar
SOD	Public	cmd	Java	?	2016 Nov
PRESTo	Public	GUI	C++	?	2016 Mar
Earthworm	Public	cmd	C	1993	2016 Nov

Table 4 Dead Seismology Software Packages

Name	Type	UI	Language	Released	Updated
rfsyn	POC	cmd	FORTRAN	2001 Mar	2001 Oct
PITSA	Public	cmd	C	?	1993 Nov
RECFUNK09	POC	cmd	FORTRAN	?	2009 Jul
Jpitsa	Public	GUI	Java	?	2000 Jan
Station Analysis Tools	POC	cmd	C	?	2015 Sept
OregonDSP	Public	lib	Java	?	2011 Feb
evalresp	Public	cmd	C	1997 Mar	2012 Oct
Seismic Handler	Private	cmd	Python	2008	2013 Feb
NonLinLoc	Public	cmd	C	?	2011 Mar
tracedsp	Private	cmd	C	?	2014 Mar
msmod	Public	cmd	C	2006	2013 Oct
FilterPicker	Public	lib	C	2011 Nov	2013 Feb
FreeUSP	Public	cmd	C	1974	2012 Apr
iaspei-tau	Public	cmd	FORTRAN	?	2009 Jun
JEvalResp	Public	cmd	Java	2003 Sep	2014 Apr
focmec	Public	cmd	FORTRAN	1984 Oct	2009 Jun
FLEXWIN	Public	cmd	C	?	2012 Aug
Mineos	Public	cmd	FORTRAN	?	2011 July
SAC	Public	cmd/lib	C	2004 Dec	2013 Nov

languages are FORTRAN (6 out of 30) and Python (2 out of 30).

Comparing Tables 3 and 4, we see that most FORTRAN software (83%) is classified as dead. Python has no examples in the dead category. The second lowest percentage of dead software is for C/C++, where only 29 % of the selected packages are dead. Java software packages have about half (56 %) of the packages classified as dead.

4.1 Product Aspect

The product aspect sums the AHP scores for installability, usability and reliability, as shown in Figure 2. The top software packages in this aspect are *JEvalResp*, *focmec*, *SOD* and *PRESTo*.

Most of the software packages have good installability, either by unzipping a compressed file, an automated installer, or a makefile. The installation guide is also provided in most cases. Some of the software products include the required external libraries as part of the zip file. This reduces the need for extra downloading by the user and ensures the use of the correct library version. Software packages like *GEE*, *CWP/SU* and *focmec* have done well by providing detailed installation guides, or an easy-to-use installer. Moreover, their installation can be verified either by an installation report or test case. However, the software *PITSA* and *OregonDSP* need improvement on installability, because we were unable to install them, due to unlisted dependencies on other libraries. The installability issues may be tied to the fact that these packages are classified as dead.

In the criteria of usability, we found most (21 out of 30) software packages provide getting started tutorials, or proper beginner examples. Many (19 out of 30) packages include explanations of the examples. Except for the unable-to-run software packages listed above, which we were unable to measure, all application have a standard “look and feel,” which contributes to their

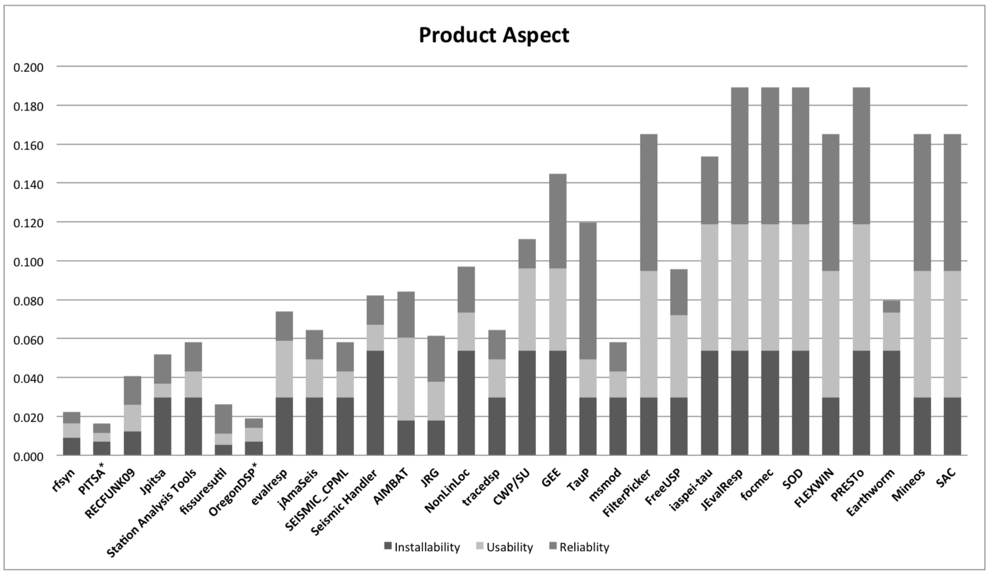

Fig. 2 Product Aspect (software marked with an asterisk (*) could not be installed)

ease of use. Nevertheless, only a few of them (3 out 30) mentioned their target users in terms of background or required knowledge. This may cause uncertainty for a new user. We were not able to find any support for several products (4 out of 30); others are supported by E-mail or community forums. Software packages like *SAC*, *FilterPicker*, *FLEXWIN* and *PRESTo* have done an obviously good job in this criteria. But others, like *PITSA*, shows its age here, with no support being provided. We were unable to find user manuals for *Jpitsa*, *rfsyn* and *fissuresutil*.

For reliability, most (25 of 30) of the software packages had no trouble with installation, but only about half (14 out of 30) provided simple initial tutorial tests, out of which 4 packages had trouble running the initial tests. We have good examples such as *SOD*, *GEE* and *FilterPicker*. However, packages like *PITSA* appear to be unreliable. *PITSA* could not be compiled and no tutorial examples were provided. Although marked as "reliable software" on its website, we found *Earthworm* had many problems while installing from source code under Linux, and the tutorial steps were difficult to understand and follow.

4.2 Implementation Aspect

In our hierarchical quality model, we have criteria of *correctness and verifiability*, *maintainability* and *understandability* under the aspect of implementation. Based on our inspection, we believe there is room of improvement for this aspect. The chart in Figure 3 provides the AHP quality assessment summary. The top software packages in this aspect are *PRESTo*, *FLEXWIN* and *FreeUSP*. The measurement of correctness for *PITSA* and *OregonDSP* may be underestimated, since these two products could not be installed.

When measuring correctness and verifiability, we found only 6 software packages had a document that could be considered as a requirement specification. No software was found that provides a formal requirement specification. Requirements documentation is important for confirming the correctness of software, since correctness can only be judged when there is a clear statement of what the tool is expected to do. Half of the software make use of standard libraries, while 5 of them are used themselves as libraries. Only half provide test data, with the other half providing standard, usually simple, examples. Only rarely is the expected output provided to verify the results. One of the good packages, *FLEXWIN*, uses *SAC* and has test cases and gives expected output. *focmec*, which is developed using *iaspei-tau*, provides test cases and expected output; in this case, the authors's paper can be considered as its informal requirement specification. We were unable to find any convincing correctness and verifiability evidence for software like *Seismic Handler* and *PITSA*.

We found that seismology software should probably pay more attention to the criterion of maintainability. Most (24 out of 30) have change logs from which we can see that major bugs have been fixed. Half of them provide a history of multiple versions. However, only a few (7 out of 30) mentioned how to contribute source code, or source code review. Moreover, only 8 of the packages have an explicit issue tracking system and only 10 packages have a version control system (5 for SVN, 5 for git). Version Control is considered crucial

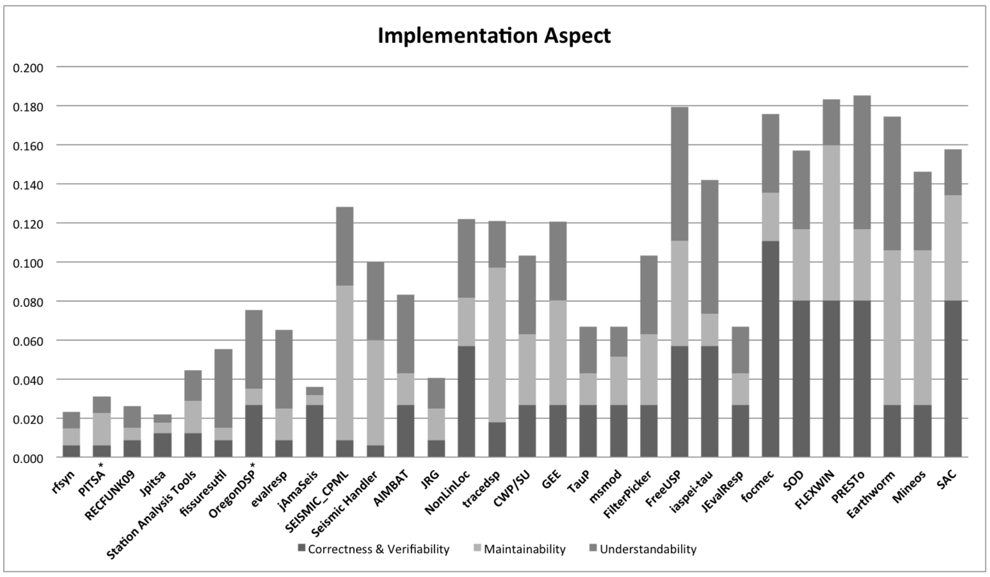

Fig. 3 Implementation Aspect (software marked with an asterisk (*) could not be installed)

tool for team collaboration (Wilson et al 2013; Wilson 2006). Software like *Mineos*, *Earthworm* and *SEISMIC_CPML* have relatively good maintainability; but *fissuresutil*, *RECFUNK09* and *Jpitsa* provide little information to convince one that maintainability was considered in their design.

Most of the seismology software packages did a good job on the criteria of understandability by having consistent coding style, meaningful identifiers and descriptive naming, etc. Nearly half of the packages (14 out of 30) gave the name or URL of the algorithm used. However, we found only a few of them (8 out of 30) had an explicitly stated coding standard. Many packages only have comments on the critical blocks of code. Unfortunately, only 3 packages have an explicit design document (including published papers). We believe a design document significantly contributes to a software's understandability. A very good example is *Earthworm*, which provides abundant information, while *PITSA* and *rfsyn* are at the other end of the understandability spectrum.

4.3 Design Aspect

For the design aspect, we examined if software qualities like robustness, performance, reusability, portability and interoperability have been considered in the design of the software packages. Figure 4 shows the AHP quality assessment for the design aspect. The top software packages in this aspect are *Mineos*, *msmod* and *SAC*. As mentioned previously, the robustness of *PITSA* and *OregonDSP* may underestimated, because they could not be installed.

During our surface check on the criteria of robustness, we checked whether each software package can gracefully handle garbage input. Since several (4 out of 30) software packages only retrieve already formatted data from network servers, they have no need to deal with different input file formats. But, there are many (12 out of 30) software packages that have no test data or are unable to run; therefore, we were not able to measure their robustness. A good example for robustness is *Taup*, since it identified incorrect input. On the other hand, *FilterPicker* failed to terminate when given an incorrectly formatted input file, although error messages were correctly displayed.

Performance was only measured on the surface because we do not have domain expertise, or enough time to measure it more deeply. To measure performance, we looked for evidence that performance had been considered in the software. There are several software packages that have performed well in this, such as the software *tracedsp*, which can take multiple input files at a time and then process them in parallel. *Mineos* used a benchmark tool to test and optimize its performance.

Based on their own designs, reusability may not be a mandatory requirement for all software, especially for standalone applications. We found nearly half of the selected packages (14 out of 30) had obvious evidence for reusability, either as library or as a tool. Many (13 out of 30) clearly documented their Application Program Interface (API), or command line tools. Some good examples, like *msmod*, can process data directly, or can be reused as a library of functions. One of the best examples of API documentation was *FilterPicker*.

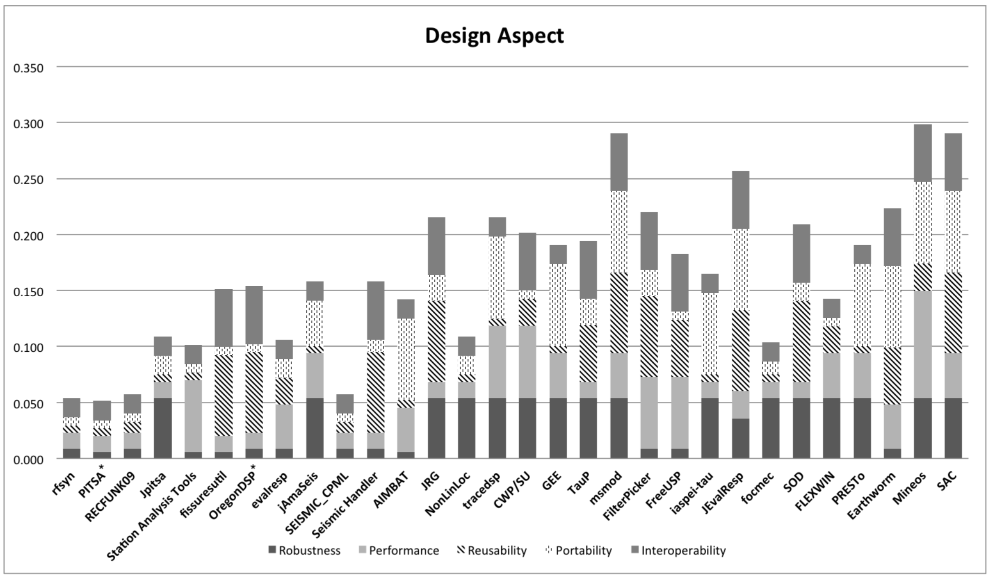

Fig. 4 Design Aspect (software marked with an asterisk (*) could not be installed)

Portability has been achieved for most of the software products. We found many of them (12 out of 30) had convincing evidence in their documentation that portability had been achieved. In fact, 19 packages have the ability to run on more than 1 platforms, including Linux, Mac and Windows. Many of them achieved portability by conditional compilation (8 out of 19) or branched versions for different operating systems (7 out of 19). The other 4 software packages achieved portability implicitly, since they are developed on platform independent languages (Python or Java).

For the criteria of interoperability, we found seismology software packages are naturally interoperable, since they generally share input or output in a standard data format. Data interchange is clearly planned in the seismology domain, and all the selected examples possess *syntactic interoperability.*

4.4 Process Aspect

The process aspect includes the criteria of visibility/ transparency, to check whether any defined development process has been used in its project, and the quality of reproducibility, which is central for long-term scientific work. Figure 5 is our quality assessment summary for the process aspect. The top software packages in this aspect are *Earthworm*, *TauP* and *FLEXWIN.*

We have stated the importance of the development process in Section 2. Unfortunately, for the criteria of visibility/ transparency, we found most of the seismology software packages have not clearly defined their development process. There is only one software package (*Earthworm*) that has explicitly specified the development process. In addition, there are 3 packages that have their source code shared and managed on Github. This decision represents a good start, as it makes progress on the development process more transparent.

For the criteria of reproducibility, not surprisingly, there is rarely any evidence that this was considered for most of the software packages. Many of them (18 out of 30) have test data, which can be considered as evidence for the reproducibility of their current version. But only a few (4 out of 30) have specified or recorded the development environment for future use, and no software uses automated tools, such as Madagascar (Fomel et al 2013), to achieve reproducibility.

5 Recommendations

In the subsections that follow recommendations are given on the design and documentation of seismology software. The advice focuses on the ideal case where the developers have the desire, time and resources to aim for high quality. That is, in the terminology of Gewaltig and Cannon (2012), the software is intended to be user ready, as opposed to review ready, or research ready. Not all developers will require a high evaluation on the grading template in Appendix A. However, if the work will be used for decision making, especially if the decisions impact safety, or if the project is to be maintained going forward, then high quality should be the goal. Moreover, if the results obtained with the software are to be reproducible, the documentation has a critical role. In the event that there are restrictions on resources, but developers want to start on the path to

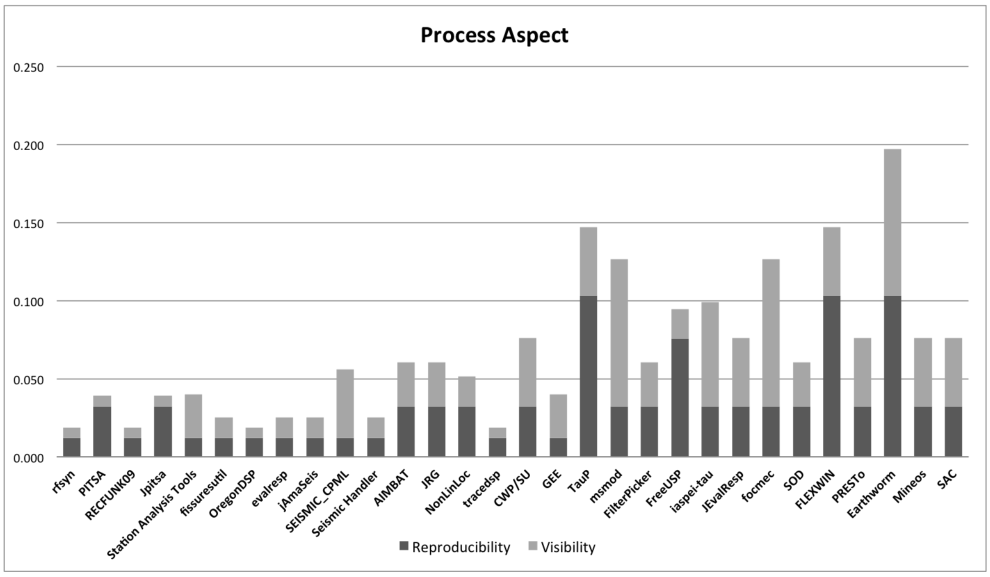

Fig. 5 Process Aspect

higher quality, the last subsection below (Section 5.10) provides advice on where and how to get started.

The paper on best practices for SC software lists 25 recommendations for individuals and groups to improve their software tools for higher productivity and reliability (Wilson et al 2013). Rather than repeat the excellent advice given therein, we will instead focus on a higher level (more abstract) style of advice. Where (Wilson et al 2013) focuses on improving the code and the use of computers to support scientific work, we will aim our advice on what documents drive quality improvement, and how to create these documents. Further details on documentation for SC software can be found in Smith (2016).

5.1 Recommended Documentation

Table 5 shows an overview of the recommended documentation for an SC project. The documents listed are typical of what is suggested for SC software certification, where certification consists of official recognition by an authority, or regulatory body, that the software is fit for its intended use. A similar set of documentation is required by the Canadian Standards Association (CSA) for quality assurance of SC programs for nuclear power plants (CSA 1999). Table 5 follows the stages of the waterfall model of software development, which may cause concern for SC practitioners, since they generally do not follow a waterfall model when developing their software (Kelly 2013; Segal 2005), and they find documenting requirements in advance challenging (Wilson et al 2013). Some developers go as far as to say that reports for each stage of software development are counterproductive (Roache 1998, p. 373). However, the reports are only counterproductive if the process used by the scientists has to follow the same waterfall as the documentation. *This most definitely does not have to be the case.* Even when the process is not waterfall, as Parnas and Clements (Parnas and Clements 1986) point out, the most logical way to present the documentation is still to "fake" a rational design process. Documentation that follows a simple rational process improves maintainability and reusability of the software artifacts.

To succeed with the qualities measured in this paper (maintainability, understandability etc.) the documentation in Table 5 should have the following qualities: complete, correct, consistent, modifiable, traceable, unambiguous and verifiable. All of these qualities are listed in the IEEE recommended practise for producing software requirements (IEEE 1998). The IEEE guidelines are for requirements, but most qualities are relevant for most documentation artifacts. In addition to the qualities listed by the IEEE, we also add that documentation should be abstract. For instance, requirements should state what is to be achieved, but be silent on how it is to be achieved.

5.2 Problem Statement

A problem statement is a high level description of what the software hopes to achieve. Like a mission statement in a strategic plan (Parker Gates 2010, p. 22), the problem statement summarizes the primary purpose of the software, what it does, who the users are and what benefits the software provides. A problem statement should be abstract. That is, it should state what the

Table 5 Recommended Documentation

Problem Statement	Description of what problem is to be solved, without mention of how to solve it.
Development Plan	Overview of development process and development infrastructure.
Requirements	Documentation of the desired functions and qualities of the software.
V & V Plan	Verification that all documentation artifacts, including the code, are internally correct. Validation, from an external viewpoint, that the right problem, or model, is being solved.
Design Specification	Documentation of how the requirements are to be realized, through both a software architecture and detailed design of modules and their interfaces.
Code	Implementation of the design in code.
V & V Report	Summary of the verification and validation efforts, including testing results.
User Manual	Instructions on how to use the software, including installation instructions and worked examples.

mission is, but not how it is to be achieved. The length of a problem statement should usually be about half a page of text, or less. The seismology software Mineos (CIG 2015) provides a good example of a problem statement on its homepage, starting with "Mineos computes synthetic seismograms in a spherically symmetric non-rotating Earth by summing normal modes." The problem statement for Mineos could benefit the wider community more if some additional context were provided. Specifically, who are the intended users and what benefit does the software provide? For instance, is Mineos unique in some way because of the algorithm it uses to calculate the seismograms?

The problem statement's main impact on software quality is through improving reuse, since a clear statement positions the current work relative to similar software products. If the problem statement shows too much overlap with existing products, the decision may be made to go in another direction. Moreover, the information in the problem statement might be enough to encourage future users and developers to adopt this product, rather than develop another one. The problem statement also improves quality since it provides focus for subsequent work and documents.

5.3 Development Plan

The recommendations in this section imply a set of documents (Table 5), but the specific contents of these documents and the process that underlies them is not prescribed. As mentioned previously, the external documentation follows a "faked" waterfall model, but the internal process can be anything. The development plan is where this internal process is specified. The specific parts of the plan should include the following: i) What documents will be created? ii) What template, including rules and guidelines, will be followed for the documents? iii) What internal process will be employed? iv) What technology infrastructure (development support tools) will be used (see Section 5.9)? v) What are the coding standards? vi) How does one contribute to the software? For the internal process there are many software development models, including spiral, agile, Rapid Application Development (RAD), etc. Developers should use what they feel comfortable with, but past software development experience has shown that ideally the process will incorporate iterative and incremental development (Larman and Basili 2003). With respect to the coding standard, a good starting point is the Java Coding Standards (Teague et al 2009).

Earthworm (Isti 2013) provides a good example of a development plan. The plan distinguishes between different categories of software: core, contributed and encapsulated. In addition, details are provided on the expected coding standards and on how to contribute to the project. Unfortunately the documentation requirements for Earthworm contributions are fairly sparse and development support tools seem to be limited to issue tracking tools. A suggestion to improve the approach to documentation would be to follow the example of the Comprehensive R Archive Network (CRAN) CRAN (2014), which facilitates a single developer contributing packages that have the quality expected of a much larger team (Smith et al 2015b). GRASS (Geographic Resources Analysis Support System) (GRASS Development Team 2014), is another example of a community developed software product with a clear software development process and an infrastructure of development support tools (Lazzarato et al 2015).

The presence of a development plan immediately improves the qualities of visibility and transparency, since the development process is now defined. Reproducibility is also improved, since the development plan includes recording development and testing details. Depending on the choices made, the development support tools, such as version control and issue tracking, can have a direct impact on the quality of maintainability.

5.4 Software Requirements Specification (SRS)

The Software Requirements Specification (SRS) records the functionality, expected performance, goals, context, design constraints, external interfaces and other quality attributes of the software (IEEE 1998). Writing an SRS generally starts with a template, which provides guidelines and rules for documenting the requirements. There are several existing templates that contain suggestions on how to avoid complications and how to improve qualities such as verifiability, maintainability and reusability (ESA February 1991; IEEE 1998; NASA 1989). There is no universally accepted template for an SRS. The recommendation here is to start with a template specifically designed for SC software (Smith et al 2007). The recommended template is suitable for SC, because of its hierarchical structure, which decomposes abstract goals to concrete instance models, with the support of data definitions, assumptions and terminology. The document's structure facilitates its maintenance and reuse (Smith and Lai 2005).

Inclusion of an SRS will directly improve several quality measures in Appendix A. For instance, the grader would find a requirements specification, a statement of user characteristics and an explicit statement on software portability. An SRS also has significant indirect benefits on other qualities, since it explicitly sets quality targets as nonfunctional requirements. For instance, usability requirements would likely improve the measures of the "look and feel" of the application and to the visibility of its features. Including an SRS improves verifiability because it provides a standard against which correctness can be judged. The recommended template (Smith and Lai 2005; Smith et al 2007) facilitates verification of the theory documented in the SRS by systematically breaking the information into structured units, and using cross-referencing. Comparing the recommended template versus an ad hoc approach, for a case study in nuclear safety analysis, highlighted several errors and omissions in the ad hoc documentation (Smith and Koothoor 2016). An SRS also indirectly improves maintainability of the software product because it improves the chance of finding errors early. A final benefit of an SRS is improved communication with stakeholders. For software to be used by others, or for others to join the development team, a clear statement of the requirements is critical.

5.5 Verification and Validation Plan/Report

Verification can be described as "solving the equations right" and validation as "solving the right equations" (Roache 1998, p. 23). SC involves using simplifying assumptions to create an idealized mathematical model of the real world. The model is then used for simulation and analysis. Verification involves checking that the governing equations for the model, together with other definitions, including boundary and/or initial conditions, are solved correctly. Validation, on the other hand, involves checking that the model is close enough to reality for whatever scientific or engineering problem is being addressed by the software. Verification is an exercise in mathematics, while validation is an exercise in engineering and science (Roache 1998, p. 24). In Roache (1998), the emphasis for verification is on the code. Here we extend the importance of verification to all software artifacts. As shown by the IEEE Standard for Software Verification and Validation Plans (van Vliet 2000, p. 412), V&V activities are recommended for each phase of the software development lifecycle.

The verification activities will tend to focus on the code, but plans should be in place for the verification of the other artifacts as well. For instance, the requirements specification should be verified by experts that can assess the reasonableness of the theoretical model, equations, assumptions etc. This verification activity is assisted by the use of a requirements template tailored to SC software, as discussed in Section 5.4. Verification of the design and the code can potentially be improved by the use of Literate Programming, as discussed in Section 5.7. An important part of the verification plan is checking the traceability between documents to ensure that every requirement is addressed by the design, every module is tested, etc.

Developing test cases is challenging for SC software, since SC problems typically lack a test oracle (Kelly et al 2011). In the absence of a test oracle, several test techniques can be used to build system tests, as described below. Many of these techniques are already in use for seismology software, but the idea here is to explicitly document the tests, so that confidence building evidence is available to all users.

– Select test cases that are a subset of the real problem for which a closed-form solution does exists.
– Build test case by assuming a solution and using this to calculate the inputs that should lead to this solution. In the case of solving Partial Differential Equations (PDEs), this approach is called the Method of Manufactured Solutions (Roache 1998).
– Compare floating point arithmetic solutions to the slower, but guaranteed correct, interval arithmetic (Hickey et al 2001).
– Compare successive grid or time step refinements.
– Compare results to another program that overlaps in functionality.
– Quantify nonfunctional requirements, like accuracy, performance and portability. Verification may include comparisons between the new implementation and competing products, since nonfunctional

requirements can naturally be stated using relative comparisons (Smith 2006).

In addition to system test cases, described above, the verification plan should outline the other testing techniques that will be used. For instance, the plan should describe how unit test cases will be selected.The test plan should also identify what, if any, code coverage metrics will be used and what approach will be employed for automated testing. If other testing techniques, such as mutation testing, or fault testing (van Vliet 2000), are to be employed, this should be included in the plan. In addition to testing, the verification plan should mention the plans for other techniques, such as code walkthroughs, code inspections, and correctness proofs (Ghezzi et al 2003; van Vliet 2000).

Although the emphasis in this description has been on verification, validation is also included in the V&V plan. For validation plans, the document should identify the real world experimental results with which the adopted physical model should agree. If the purpose of the code is a general purpose mathematical library, such as a library for signal processing, there may not be a need for a separate validation phase, since the code is not directly tied to a model of the real world.

Having a V&V plan will improve the quality measures outlined in this paper. For instance, installability should be improved through an explicit plan for testing the installation process. Similarly, performance and portability tests will be outlined in the V&V plan. A V&V plan improves the assessment of correctness and verifiability, since the plan explicitly identifies the tests and other techniques used to improve these qualities. Moreover, a consequence of the emphasis on verification should be an improvement in reliability. In the case where the V&V plan suggests code inspections, there should be a corresponding improvement in understandability, due to an increase in the number of people reading the code. Finally, a V&V plan improves reproducibility, since the plan will record the environment for development and testing, as well as providing test data. Ideally the V&V plan will also facilitate an automated testing infrastructure, which will in turn improve reproducibility.

The V&V report summarizes the test results, with enough detail to convince a reader that all the planned activities were accomplished. The report should emphasize changes made in a response to uncovered issues.

5.6 Design Specification

As mentioned in Section 4.2, only 3 of the 30 surveyed packages have explicit design documentation. The absence of documentation effects design, implementation and maintenance (Hoffman and Strooper 1995, p. 16). The recommended approach to handle complexity in design is to use abstraction (van Vliet 2000, p. 296). For scientific software, the inspiration for the appropriate abstraction should usually be taken from the underlying mathematics.

The recommended documentation should include a high level view of the software architecture, which divides the system into modules, and a low level view, which specifies the interfaces for the modules. In this context, a module is defined as a “work assignment given to a programmer or group of programmers” (Parnas et al 1984). Wilson et al (2013) advise modular design for SC, but are silent on the criteria to use to decompose the software into modules. We advocate a decomposition based on the principle of information hiding (Parnas 1972). This principle supports design for change, because the “secrets” that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored. The modular decomposition can be recorded in a Module Guide (MG) (Parnas et al 1984), which organizes the modules in a hierarchy by their secrets. An SC example of a parallel mesh generator, which follows the Parnas approach to information hiding, can be found in (Smith and Yu 2009).

The MG does not provide enough information for each module to be developed. The interface still needs to be designed and documented in a Module Interface Specification (MIS) document (Hoffman and Strooper 1995). The MIS is less abstract than the architectural design mentioned above. However, an MIS is still abstract, since it describes what the module will do, but not how to do it. The interfaces can be documented formally (ElSheikh et al 2004; Smith and Yu 2009) or informally. An informal presentation would use natural language and equations. The most important concern is that the specification needs to clearly define all parameters, since an unclear description of the parameters is one cause of reusability issues for libraries (Dubois 2002). The designer should keep in mind the following interface quality criteria: consistent, essential, general, minimal and opaque (Hoffman and Strooper 1995, p. 83).

Documentation of the system architecture and the module interfaces will improve the software quality. For instance, the documentation provides evidence that maintainability has been considered, since the design will be based on likely changes. The documentation of the API (through an MIS) improves the measures associated with reusability and interoperability. Furthermore, the documentation improves understandability, since the code is modularized, the design is documented, and the parameter ordering will be standardized.

5.7 Code

We recommend reusing mature and trustworthy libraries when possible. Using mature libraries saves development time and reduces errors. Systems can be developed faster if they can be built on *stable subsystems* (Simon 1996). Moreover, reuse supports maintenance through software *evolution* (Dawkins 1996; Fischer 2001).

For any of the code that is written, we recommend that comments be given the attention that they deserve. Comments "aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings" (Kernighan and Pike 1999). Comments should not describe details of how an algorithm is implemented, but instead focus on what the algorithm does and the strategy behind it. Good examples of commented code from seismology include Seismic Handler and OregonDSP. Writing comments is one of the best practices identified for SC by (Wilson et al 2013). As said by Wilson et al., we should aim to "write programs for people, not computers" and "[t]he best way to create and maintain reference documentation is to embed the documentation for a piece of software in that software" (Wilson et al 2013). An approach that takes these ideas to their logical conclusion is Literate Programming (LP) (Knuth 1984).

LP was introduced by Knuth (1983). "...[I]nstead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do" (Knuth 1992, pg. 99). When using LP, an algorithm is refined into smaller, simpler parts. Each of the parts is documented in an order that is natural for human comprehension, as opposed to the order used for compilation. In a literate program, documentation and code are maintained in one source. The program is written as an interconnected "web" of "chunks" of code (Knuth 1983, 1992). LP can improve verifiability, understandability and reproducibility. One example of a commonly used LP tools is Sweave, which is part of the tool suite used by CRAN (Comprehensive R Archive Network). However, the aim of Sweave is teaching the use of the code to new users, as opposed to convincing other developers that the implementation is correct (Smith et al 2015b). Other examples that place a greater emphasis on verifiability for SC software can be found in Nedialkov (2010) and Pharr and Humphreys (2004). Specific documentation and code examples that highlight the quality improvements possible using LP can be found in Smith and Koothoor (2016).

Library reuse and code comments will improve the software quality. For instance, correctness and verifiability are improved by the presence of trustworthy external libraries. These qualities are further improved if a confidence building technique, such as LP, is used. LP also provides evidence that the project is concerned about reusability and maintainability, since proper documentation means that others can reuse and improve the existing work. Paying proper attention to comments improves understandability, which is further improved if one follows the other coding conventions mentioned under this quality in Appendix A.

5.8 User Manual

The presence of a user manual will have a direct impact on quality. The quality of usability in particular benefits from a user manual, especially if the manual includes a getting started tutorial and a fully explained example. A user manual also benefits installability, as long as it includes linear installation instructions. In addition to the components of the manual explicitly checked in the grading template, a user manual will typically also include the following sections: system requirements, instructions for use, troubleshooting, frequently asked questions, and safety precautions (if appropriate). Advice for writing user manuals and technical instructions can be found in technical writing texts, such as Blicq (1987) and VanAlstyne (2005).

We found several good sample manuals, such as those for GEE and Mineos. Outside of seismology, the CRAN repository provides consistently high usability (Smith et al 2015b). Thanks to the CRAN repository policy, Rd files and Sweave vignettes, the R extensions tend to provide complete and consistent user documentation. The CRAN policy, together with the support tools, allows a single developer project to have the quality of something developed by a larger team (Smith et al 2015b). With respect to usability, a notably good CRAN example is mokken (van der Ark 2013).

5.9 Tool Support

We recommend software development tools for issue tracking and version control. As shown in Section 4.2, the use of these tools is rare for seismology software. Issue tracking is considered to be a central quality assurance process (Neumann 2009). One option is to use a commercial issue tracker, such as Jira. Free tools are also available, such as, iTracker, Roundup, GitHub and

Bugzilla (Johnson and Dubois 2003). For version control, the recommended tools are SVN (Collins-Sussman et al 2004) and Git (Loeliger and McCullough 2012). The model employed by SVN is a central repository. Git, on the other hand, uses distributed version control, which makes it more flexible than SVN. Version control tools can also be used for reproducibility, since they are able to record development information as the project progresses. However, Davison (Davison 2012), recommends more flexible and powerful automated reproducibility tools, such as Sumatra (Davison 2013) and Madagascar (Fomel et al 2013).

Tool use for code documentation falls on a continuum between no tool use, all the way up to full Literate Programming (discussed in Section 5.7). From our survey, seismology software code documentation ranges from "no tool" up to code documentation assistants, which encourage completeness and consistency. The tools in this later category include Javadoc, Doxygen and Sphinx. For code written with Matlab, the publish function sits at a similar level on the tool use continuum. For seismology software, we saw Javadoc in use by *JEvalResp*, *fissuresutil*, *TauP*, *SOD* and *jAmaSeis*; Doxygen is employed by *Earthworm*. The previously mentioned tools can be thought of as code first, then documentation. LP flips this around with documentation first, then code. Tools for LP include cweb, noweb, FunnelWeb and Sweave. We did not see any evidence of LP use for seismology software.

Tools also exist to make the testing phase easier. For functional testing, unit testing frameworks are very popular. A unit testing framework, has been developed for most programming languages. Examples for the languages listed in Tables 3 and 4 include JUnit (for Java), Cppunit (for C++), CUnit (for C), FUnit (for FORTRAN), and PyUnit (for Python). The use of a unit testing framework for the surveyed seismology software is limited, with only *TauP* and *fissuresutil* using such a framework (JUnit). For nonfunctional testing related to performance, one can use a profiler to identify the real bottlenecks in performance (Wilson et al 2013). A powerful tool for dynamic analysis of code is Valgrind.

5.10 Practical Advice

Smith et al (2016a) showed that the style of documentation recommended above has value for SC developers. However, that study also showed that document driven design is a daunting task to start out with, especially for projects that begin with a small scope. As a starting point, we provide some practical "getting started" advice. To start with, we recommend using the template in Appendix A as a checklist for reviewing a project as it progresses. Of particular importance for the initial efforts is the quality of installability. Several of the seismology projects in this study could not be installed. If others have the same problem, then it means that nobody will use the software. Another practical recommendation is to use tools wherever possible to simplify and improve the development process. In particular, a version control system is an important building block (Wilson 2006). We recommend adopting a full web solution, like GitHub, or SourceForge, which provide documentation and code management, along with issue tracking. This approach provides the advantage that the product website can be designed for maximum visibility (Lazzarato et al 2015). Moreover, the project can gradually grow into the use of the tools that are available as the need arises. For code documentation, for seismology software we recommend using a tool like Doxygen, since this enforces some consistency and produces documentation that other developers can more easily navigate than the source code alone.

Although we place a great deal of importance on requirements, testing is often a more natural starting point, likely because tests are less abstract than requirements. We recommend beginning the process by writing test cases, which in a sense form the initial requirements for the project. If an infrastructure for automated testing is created early in a project, this can help improve verification and validation efforts going forward.

6 Conclusion

Our summary and recommendations are based on the assumption that the software that we found available on-line is intended to be user ready. However, in the terminology of Gewaltig and Cannon (2012), some projects may only need to reach the level of being review ready, or research ready. For a small team of developers working on specialized software that is not intended to have a long life, or be maintainable going forward, the summary comments and recommendations do not apply.

Our study suggests that seismology software follows the usual trends for SC software. For instance, as for other SC software, requirements documentation is not emphasized (Carver et al 2007), nor is there much evidence of efficient testing (Sanders and Kelly 2008; Segal 2007b). We assume that many seismologists are following the usual pattern of treating software development as a "secondary activity" to their research job (Segal 2007b). By paying a little more attention to software engineering, we believe that overall seismology software quality can be improved and software development productivity can be increased.

The above statements on the state of seismology software development are based data collected on 30 seismology packages. We used a hierarchical quality model, starting from four aspects: product, implementation, design and process. Our overall process is similar to that used for other studies of SC software (Smith et al 2016b; Lazzarato et al 2015; Smith et al 2015a,b,c). Each software product is graded following a standard template, consisting of 56 questions. Following this, we construct AHP tables to build a relative quality comparison between products.

We found that most of the seismology software could likely benefit from standardizing their development process. The frequent lack of a requirement specification documents, test data, assistant tools and tutorial examples is a problem that would likely be worth addressing. The highlights of our findings are as follows:

- Summing the contributions to all four aspects shows the top three seismology packages as *SAC*, *Mineos* and *Earthworm*.
- The programming languages used are C, C++, FORTRAN, Python and Java.
- For the product aspect (installability, usability and reliability) the top products are *JEvalResp*, *focmec*, *SOD* and *PRESTo*.
- For the implementation aspect (correctness and verifiability, maintainability and understandability) the top programs are *PRESTo*, *FLEXWIN* and *FreeUSP*.
- Only 6 of the 30 packages had a document that could be considered as a requirements specification.
- Only half of the packages provide test data and only rarely is the expected output stated.
- Only 8 packages have an explicitly issue tracking system and only 10 packages have a version control system.
- Only 3 programs have an explicit design document.
- For the design aspect (robustness, performance, reusability, portability and interoperability) the top software packages are *Mineos*, *msmod* and *SAC*.
- For the criteria of interoperability, we found seismology software packages are naturally interoperable.
- For the process aspect (visibility/transparency, reproducibility) the top performers are *Earthworm*, *TauP* and *FLEXWIN*.
- Only one package (*Earthworm*) explicitly specified the development process used.
- For reproducibility only 4 out of 30 packages specified or recorded the development environment for future use.

Our recommendation for addressing these problems is to follow a document driven process (Smith and Yu 2009; Smith and Koothoor 2016). This process includes a problem statement, development plan, requirements specification, V&V plan, design specification, code, V&V report and a user manual. We also provide advice on tool use, including issue tracking, version control, code documentation and testing tools. As a practical starting point for quality improvement, we recommend documenting tests cases as requirements, using GitHub (or SourceForge), incorporating Doxygen and implementing automated testing via a unit testing framework.

Acknowledgements Adam Lazzarato, Yue Sun and Vasudha Kapil are acknowledged for their support on this project. Constructive criticism on an earlier draft provided by Dr. Diane Kelly is gratefully acknowledged.

References

Almeida JB (2011) Rigorous Software Development. Springer London Dordrecht Heidelberg New York

van der Ark LA (2013) mokken: Mokken Scale Analysis in R. URL `http://cran.r-project.org/web/packages/mokken/index.html`, R package version 2.7.5

Bidoit M, Sannella D, Tarlecki A (2002) Architectural specifications in casl. Formal Aspects of Computing 13(3-5):252–273

Blicq R (1987) Technically-Write! Prentice Hall

Carver JC, Kendall RP, Squires SE, Post DE (2007) Software development environments for scientific and engineering software: A series of case studies. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, pp 550–559, DOI http://dx.doi.org/10.1109/ICSE.2007.77

CIG (2015) Mineos. `http://geodynamics.org/cig/software/mineos/`

Collins-Sussman B, Fitzpatrick B, Pilato M (2004) Version Control with Subversion. O'Reilly Media, Inc.

CRAN (2014) The comprehensive r archive network. `http://cran.r-project.org/`, URL `http://cran.r-project.org/`

CSA (1999) Quality assurance of analytical, scientific, and design computer programs for nuclear power plants. Tech. Rep. N286.7-99, Canadian Standards Association, 178 Rexdale Blvd. Etobicoke, Ontario, Canada M9W 1R3

Davison AP (2012) Automated capture of experiment context for easier reproducibility in computational research. Computing in Science and Engineering, IEEE

Davison AP (2013) Sumatra Project Main Page. `http://neuralensemble.org/sumatra/`

Dawkins R (1996) The blind watchmaker: Why the evidence of evolution reveals a universe without design. WW Norton & Company

Dubois PF (2002) Designing scientific components. Computing in Science and Engineering 4(5):84–90

ElSheikh AH, Smith WS, Chidiac SE (2004) Semi-formal design of reliable mesh generation systems. Advances in Engineering Software 35(12):827–841

ESA (February 1991) ESA software engineering standards, PSS-05-0 issue 2. Tech. rep., European Space Agency

Fischer G (2001) The software technology of the 21st century: From software reuse to collaborative software design. In: International Symposium on Future Software Technology, pp 1–8

Fomel S, Sava P, Vlad I, Liu Y, Bashkardin V (2013) Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments. Journal of Open Research Software 1(1):e8, DOI http://dx.doi.org/10.5334/jors.ag

Forman EH, Gass SI (2001) The analytic hierarchy process—an exposition. Operations Research 49(4):469–486

for Geodynamics CI (2014) List of Software. `http://geodynamics.org/cig/software/`

Gewaltig MO, Cannon R (2012) Quality and sustainability of software tools in neuroscience. Cornell University Library arXiv preprint arXiv:1205.3025:20 pp

Gewaltig MO, Cannon R (2014) Current practice in software development for computational neuroscience and how to improve it. PLoS computational biology 10(1)

Ghezzi C, Jazayeri M, Mandrioli D (2003) Fundamentals of Software Engineering, 2nd edn. Prentice Hall, Upper Saddle River, NJ, USA

Gillies A (2011) Software Quality: Theory and Management. Lulu. com, URL `http://books.google.ca/books?id=XTvpAQAAQBAJ`

GRASS Development Team (2014) GRASS GIS bringing advanced geospatial technologies to the world. `http://grass.osgeo.org/`, URL `http://grass.osgeo.org/`

Hickey T, Ju Q, Van Emden MH (2001) Interval arithmetic: From principles to implementation. J ACM 48(5):1038–1068, DOI 10.1145/502102.502106, URL `http://doi.acm.org/10.1145/502102.502106`

Hoffman DM, Strooper PA (1995) Software Design, Automated Testing, and Maintenance: A Practical Approach. International Thomson Computer Press, URL `http://citeseer.ist.psu.edu/428727.html`

IEEE (1990) IEEE Standard Glossary of Software Engineering Terminology. Tech. rep., Institute of Electronic and Electrical Engineers (IEEE), DOI 10.1109/ieeestd.1990.101064, URL `http://dx.doi.org/10.1109/ieeestd.1990.101064`

IEEE (1998) Recommended practice for software requirements specifications. IEEE Std 830-1998 pp 1–40, DOI 10.1109/IEEESTD.1998.88286

ISO/IEC (2010) ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Tech. rep., International Organization for Standardization (ISO)

Isti (2013) Earthworm software standards. `http://www.earthwormcentral.org/documentation2/PROGRAMMER/SoftwareStandards.html`

Johnson JN, Dubois PF (2003) Issue tracking. Computing in Science & Engineering 5(6):71–77

Kelly DF (2007) A software chasm: Software engineering and scientific computing. IEEE Software 24(6):120–119, DOI http://dx.doi.org/10.1109/MS.2007.155

Kelly DF (2013) Industrial scientific software: A set of interviews on software development. In: Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp., Riverton, NJ, USA, CASCON '13, pp 299–310, URL `http://dl.acm.org/citation.cfm?id=2555523.2555555`

Kelly DF, Smith WS, Meng N (2011) Software engineering for scientists. Computing in Science & Engineering 13(5):7–11

Kernighan BW, Pike R (1999) The practice of programming. Addison-Wesley Professional

Knuth DE (1983) The WEB system of structured documentation. Stanford Computer Science Report CS980, Stanford University, Stanford, CA

Knuth DE (1984) Literate programming. The Computer Journal 27(2):97–111, DOI 10.1093/comjnl/27.2.97, URL `http://comjnl.oxfordjournals.org/content/27/2/97.abstract`, `http://comjnl.oxfordjournals.org/content/27/2/97.full.pdf+html`

Knuth DE (1992) Literate Programming. CSLI Lecture Notes Number 27, Center for the Study of Language and Information, URL `http://csli-www.stanford.edu/publications/literate.html`

Lang JP (2013) IRIS SeisCode. `https://seiscode.iris.washington.edu`

Larman C, Basili VR (2003) Iterative and incremental development: A brief history. Computer 36(6):47–56

Lazzarato A, Smith WS, Carette J (2015) State of the practice for remote sensing software. Technical Report CAS-15-03-SS, McMaster University

Loeliger J, McCullough M (2012) Version Control with Git: Powerful tools and techniques for collaborative software development. " O'Reilly Media, Inc."

NASA (1989) Software requirements DID, SMAP-DID-P200-SW, release 4.3. Tech. rep., National Aeronautics and Space Agency

Nedialkov NS (2010) Implementing a Rigorous ODE Solver through Literate Programming. Tech. Rep. CAS-10-02-NN, Department of Computing and Software, McMaster University

Neumann D (2009) The impact of communication structure on issue tracking efficiency at a large business software vendor. Issues in Information Systems X(2):316–323

Parker Gates L (2010) Strategic planning with critical success factors and future scenarios: An integrated strategic planning strategic planning with critical success factors and future scenarios: An integrated strategic planning framework. Tech. Rep. CMU/SEI-2010-TR-037, Software Engineering Institute, Carnegie-Mellon University

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Comm ACM, vol 15, no 2, pp 1053-1058

Parnas DL, Clements P (1986) A rational design process: How and why to fake it. IEEE Transactions on Software Engineering 12(2):251–257

Parnas DL, Clement PC, Weiss DM (1984) The modular structure of complex systems. In: International Conference on Software Engineering, pp 408–419

Pharr M, Humphreys G (2004) Physically Based Rendering: From Theory to Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA

Roache PJ (1998) Verification and Validation in Computational Science and Engineering. Hermosa Publishers, Albuquerque, New Mexico

Saaty TL (1990) How to make a decision: the analytic hierarchy process. European journal of operational research 48(1):9–26

Saaty TL (2008-09-01) Relative measurement and its generalization in decision making why pairwise comparisons are central in mathematics for the measurement of intangible factors the analytic hierarchy/network process. RACSAM - Revista de la Real Academia de Ciencias Exactas, Fisicas y Naturales Serie A Matematicas 102:251–318, DOI 10.1007/BF03191825

Sanders R, Kelly DF (2008) Dealing with risk in scientific software development. IEEE Software 25:21

Segal J (2005) When software engineers met research scientists: A case study. Empirical Software Engineering 10(4):517–536, DOI 10.1007/s10664-005-3865-y, URL http://dx.doi.org/10.1007/s10664-005-3865-y

Segal J (2007a) End-user software engineering and professional end-user developers. In: Burnett MH, Engels G, Myers BA, Rothermel G (eds) End-User Software Engineering, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, no. 07081 in Dagstuhl Seminar Proceedings, URL http://drops.dagstuhl.de/opus/volltexte/2007/1095

Segal J (2007b) Some problems of professional end user developers. Visual Languages and Human-Centric Computing, 2007 VL/HCC 2007 IEEE Symposium on pp 111 – 118

Segal J, Morris C (2008) Developing scientific software. IEEE Software 25(4):18–20

Simmons S, Sea I (1994) Software design for installability. USENIX

Simon HA (1996) The sciences of the artificial. 3rd

Smith WS (2006) Systematic development of requirements documentation for general purpose scientific computing software. IEEE International Requirements Engineering Conference

Smith WS (2016) A rational document driven design process for scientific computing software. In: Carver JC, Hong NC, Thiruvathukal G (eds) Software Engineering for Science, Chapman & Hall/CRC Computational Science, Taylor & Francis, chap Examples of the Application of Traditional Software Engineering Practices to Science, pp 33–63

Smith WS, Koothoor N (2016) A document-driven method for certifying scientific computing software for use in nuclear safety analysis. Nuclear Engineering and Technology 48(2):404–418, DOI http://dx.doi.org/10.1016/j.net.2015.11.008, URL http://www.sciencedirect.com/science/article/pii/S1738573315002582

Smith WS, Lai L (2005) A new requirements template for scientific computing. In: Ralyté J, Ågerfalk P, Kraiem N (eds) Proceedings of the First International Workshop on Situational Requirements Engineering Processes – Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP'05, In conjunction with 13th IEEE International Requirements Engineering Conference, Paris, France, pp 107–121

Smith WS, Yu W (2009) A document driven methodology for improving the quality of a parallel mesh generation toolbox. Advances in Engineering Software 40(11):1155–1167, DOI http://dx.doi.org/10.1016/j.advengsoft.2009.05.003

Smith WS, Lai L, Khedri R (2007) Requirements analysis for engineering computation: A systematic approach for improving software reliability. Reliable Computing, Special Issue on Reliable Engineering Computation 13(1):83–107

Smith WS, Lazzarato A, Carette J (2015a) State of the practice for mesh generation software. Technical Report CAS-15-04-SS, McMaster University

Smith WS, Sun Y, Carette J (2015b) Comparing psychometrics software development between CRAN and other communities. Technical Report CAS-15-01-SS, McMaster University

Smith WS, Sun Y, Carette J (2015c) State of the practice for developing oceanographic software. Technical Report CAS-15-02-SS, McMaster University, Department of Computing and Software

Smith WS, Jegatheesan T, Kelly DF (2016a) Advantages, disadvantages and misunderstandings about document driven design for scientific software. In: Proceedings of the Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCE), 8 pp

Smith WS, Lazzarato A, Carette J (2016b) State of practice for mesh generation software. Advances in Engineering Software 100:53–71

Teague LD, Somannair R, Stark K, Goyal N, Ruzbacki R (2009) Java coding standards, Software Engineering Standards Division, Department of Veterans Affairs. www.va.gov/trm/files/java_coding_standards_v2.doc

Triantaphyllou SHM (1995) Using the analytic hierarchy process for decision making in engineering applications. International Journal of Industrial Engineering: Applications and Practice 2(1):35–44

VanAlstyne JS (2005) Professional and Technical Writing Strategies, sixth edn. Pearson Prentice Hall, Upper Saddle River, New Jersey

van Vliet H (2000) Software Engineering (2nd ed.): Principles and Practice. John Wiley & Sons, Inc., New York, NY, USA

Wilson GV (2006) Where's the real bottleneck in scientific computing? American Scientist 94(1):5

Wilson GV, Aruliah D, Brown CT, Hong NPC, Davis M, Guy RT, Haddock SH, Huff KD, Mitchell IM, Plumblet MD, Waugh B, White EP, Wilson P (2013) Best practices for scientific computing. CoRR abs/1210.0530

A Full Grading Template

The table below lists the full set of measures that are assessed for each software product. The measures are grouped under headings for each quality, and one for summary information. Following each measure, the type for a valid result is given in brackets. Many of the types are given as enumerated sets. For instance, the response on many of the questions is one of "yes," "no," or "unclear." The type "number" means natural number, a positive integer. The types for date and url are not explicitly defined, but they are what one would expect from their names. In some cases the response for a given question is not necessarily limited to one answer, such as the question on what platforms are supported by the software product. Case like this are indicated by "set of" preceding the type of an individual answer. The type in these cases are then the power set of the individual response type. In some cases a superscript * is used to indicate that a response of this type should be accompanied by explanatory text. For instance, if problems were caused by uninstall, the reviewer should note what problems were caused. An (I) precedes the question description when its measurement requires a successful installation.

B Summary of Grading Results

The full gradings of the 30 RS software products start on the next page. The most recent gradings are available at: `http://dx.doi.org/10.17632/67ncspgz8n.1`. The column headings correspond with the above questions from the grading template.

Summary Information
Software name? (string)
URL? (url)
Educational institution (string)
Software purpose (string)
Number of developers (number)
How is the project funded (string)
Number of downloads for current version (number)
Release date (date)
Last updated (date)
Status ({alive, dead, unclear})
License ({GNU GPL, BSD, MIT, terms of use, trial, none, unclear})
Platforms (set of {Windows, Linux, OS X, Android, Other OS})
Category ({concept, public, private})
Development model ({open source, freeware, commercial})
Publications using the software (set of url)
Publications about the software (set of url)
Is source code available? ({yes, no})
Programming language(s) (set of {FORTRAN, Matlab, C, C++, Java, R, Ruby, Python, Cython, BASIC, Pascal, IDL, unclear})

Installability (Measured via installation on a virtual machine.)
Are there installation instructions? ({yes, no})
Are the installation instructions linear? ({yes, no, n/a})
Is there something in place to automate the installation? ({yes*, no})
Is there a specified way to validate the installation, such as a test suite? ({yes*, no})
How many steps were involved in the installation? (number)
How many software packages need to be installed before or during installation? (number)
(I) Run uninstall, if available. Were any obvious problems caused? ({unavail, yes*, no})
Overall impression? ({1 .. 10})

Correctness and Verifiability
Are external libraries used? ({yes*, no, unclear})
Does the community have confidence in this library? ({yes, no, unclear})
Any reference to the requirements specifications of the program? ({yes*, no, unclear})
What tools or techniques are used to build confidence of correctness? (string)
(I) If there is a getting started tutorial, is the output as expected? ({yes, no*, n/a})
Overall impression? ({1 .. 10})

Surface Reliability
Did the software "break" during installation? ({yes*, no})
(I) Did the software "break" during the initial tutorial testing? ({yes*, no, n/a})
Overall impression? ({1 .. 10})

Surface Robustness
(I) Does the software handle garbage input reasonably? ({yes, no*})
(I) For any plain text input files, if all new lines are replaced with new lines and carriage returns, will the software handle this gracefully? ({yes, no*, n/a})
Overall impression? ({1 .. 10})

Surface Performance
Is there evidence that performance was considered? ({yes*, no})
Overall impression? ({1 .. 10})

Surface Usability
Is there a getting started tutorial? ({yes, no})
Is there a standard example that is explained? ({yes, no})
Is there a user manual? ({yes, no})
(I) Does the application have the usual “look and feel” for the platform it is on? ({yes, no*})
(I) Are there any features that show a lack of visibility? ({yes, no*})
Are expected user characteristics documented? ({yes, no})
What is the user support model? (string)
Overall impression? ({1 .. 10})

Maintainability
Is there a history of multiple versions of the software? ({yes, no, unclear})
Is there any information on how code is reviewed, or how to contribute? ({yes*, no})
Is there a changelog? ({yes, no})
What is the maintenance type? (set of {corrective, adaptive, perfective, unclear})
What issue tracking tool is employed? (set of {Trac, JIRA, Redmine, e-mail, discussion board, sourceforge, google code, git, none, unclear})
Are the majority of identified bugs fixed? ({yes, no*, unclear})
Which version control system is in use? ({svn, cvs, git, github, unclear})
Is there evidence that maintainability was considered in the design? ({yes*, no})
Are there code clones? ({yes*, no, unclear})
Overall impression? ({1 .. 10})

Reusability
Are any portions of the software used by another package? ({yes*, no})
Is there evidence that reusability was considered in the design? (API documented, web service, command line tools, ...) ({yes*, no, unclear})
Overall impression? ({1 .. 10})

Portability
What platforms is the software advertised to work on? (set of {Windows, Linux, OS X, Android, Other OS})
Are special steps taken in the source code to handle portability? ({yes*, no, n/a})
Is portability explicitly identified as NOT being important? ({yes, no})
Convincing evidence that portability has been achieved? ({yes*, no})
Overall impression? ({1 .. 10})

Surface Understandability (Based on 10 random source files)
Consistent indentation and formatting style? ({yes, no, n/a})
Explicit identification of a coding standard? ({yes*, no, n/a})
Are the code identifiers consistent, distinctive, and meaningful? ({yes, no*, n/a})
Are constants (other than 0 and 1) hard coded into the program? ({yes, no*, n/a})
Comments are clear, indicate what is being done, not how? ({yes, no*, n/a})
Is the name/URL of any algorithms used mentioned? ({yes, no*, n/a})
Parameters are in the same order for all functions? ({yes, no*, n/a})
Is code modularized? ({yes, no*, n/a})
Descriptive names of source code files? ({yes, no*, n/a})
Is a design document provided? ({yes*, no, n/a})
Overall impression? ({1 .. 10})

Interoperability
Does the software interoperate with external systems? ({yes*, no})
Is there a workflow that uses other softwares? ({yes*, no})
If there are external interactions, is the API clearly defined? ({yes*, no, n/a})
Overall impression? ({1 .. 10})

Visibility/Transparency
Is the development process defined? If yes, what process is used. ({yes*, no, n/a})
Ease of external examination relative to other products considered? ({1 .. 10})
Overall impression? ({1 .. 10})

Reproducibility
Is there a record of the environment used for their development and testing? ({yes*, no})
Is test data available for verification? ({yes, no})
Are automated tools used to capture experimental context? ({yes*, no})
Overall impression? ({1 .. 10})

Table 6 Full List of Seismology Software

Name	URL	AHP Score
rfsyn	`https://seiscode.iris.washington.edu/projects/rfsyn`	0.009
PITSA	`https://seiscode.iris.washington.edu/projects/pitsa`	0.011
RECFUNK09	`https://seiscode.iris.washington.edu/projects/recfunk09-pick`	0.011
Jpitsa	`http://www.iris.edu/pub/programs/JPITSA/`	0.017
Station Analysis Tools	`https://seiscode.iris.washington.edu/projects/station-analysis-tools`	0.019
fissuresutil	`https://github.com/crotwell/fissuresImpl`	0.020
OregonDSP	`https://seiscode.iris.washington.edu/projects/oregondsp`	0.021
evalresp	`http://www.iris.edu/dms/nodes/dmc/software/downloads/evalresp/3-3-3/`	0.021
jAmaSeis	`http://www.iris.edu/hq/jamaseis/`	0.022
SEISMIC_CPML	`https://github.com/geodynamics/seismic_cpml`	0.023
Seismic Handler	`http://www.seismic-handler.org/wiki`	0.028
AIMBAT	`https://github.com/pysmo/aimbat`	0.028
JRG	`http://crack.seismo.unr.edu/jrg/`	0.029
NonLinLoc	`http://alomax.free.fr/nlloc/`	0.029
tracedsp	`https://seiscode.iris.washington.edu/projects/tracedsp`	0.032
CWP/SU	`http://www.cwp.mines.edu/cwpcodes/`	0.038
GEE	`http://www.seis.sc.edu/gee/`	0.038
TauP	`http://www.seis.sc.edu/taup/index.html`	0.041
msmod	`https://seiscode.iris.washington.edu/projects/msmod`	0.042
FilterPicker	`http://alomax.free.fr/FilterPicker/`	0.042
FreeUSP	`http://freeusp.org/FreeUSP.html`	0.042
iaspei-tau	`http://www.iris.edu/pub/programs/iaspei-tau/`	0.043
JEvalResp	`http://www.iris.edu/pub/programs/JEvalResp/JEvalResp_v1.77/`	0.045
focmec	`http://www.iris.edu/pub/programs/focmec/`	0.046
SOD	`http://www.seis.sc.edu/sod/`	0.047
FLEXWIN	`http://geodynamics.org/cig/software/flexwin/`	0.049
PRESTo	`http://www.prestoews.org`	0.049
Earthworm	`http://www.earthwormcentral.org/`	0.052
Mineos	`http://geodynamics.org/cig/software/mineos/`	0.053
SAC	`http://www.iris.edu/dms/nodes/dmc/software/downloads/sac/`	0.053

Table 7 Installability, II: Installation instructions available, II linear: Linear installation steps, AI: Automated Installation, Inst ValidTests for installation validation, # S/W lib: Number of software/libraries required for installation, Uninst: Any uninstallation problem?

Name	II	II linear	AI	Inst Valid	# of Steps	# S/w lib	Uninst
rfsyn	Yes	Yes	No	No	2	0	No
PITSA	Yes	No	Yes	No	2	0	No
RECFUNK09	Yes	Yes	N/A	No	1	2	No
Jpitsa	Yes	Yes	Yes	No	1	0	No
Station Analysis Tools	Yes	Yes	Yes	Yes	5	Unclear	No
fissuresutil	N/A	N/A	N/A	N/A	1	0	No
OregonDSP	No	Yes	N/A	No	N/A	0	No
evalresp	Yes	Yes	Yes	No	2	0	No
jAmaSeis	Yes	Yes	Yes	No	1	0	No
SEISMIC_CPML	Yes	Yes	Yes	No	1	0	No
Seismic Handler	Yes	Yes	Yes	No	1	4	No
AIMBAT	Yes	Yes	N/A	Yes	2	4	No
JRG	Yes	Yes	No	Yes	1	1	No
NonLinLoc	Yes	Yes	Yes	Yes	1	0	No
tracedsp	Yes	Yes	Yes	No	1	0	No
CWP/SU	Yes	Yes	Yes	Yes	7	0	No
GEE	Yes	Yes	Yes	Yes	1	0	No
TauP	Yes	Yes	No	Yes	2	1	No
msmod	Yes	Yes	Yes	No	1	0	No
FilterPicker	Yes	Yes	Yes	Yes	2	0	No
FreeUSP	Yes	Yes	Yes	No	3	0	No
iaspei-tau	Yes	Yes	Yes	Yes	2	0	No
JEvalResp	Yes	Yes	Yes	Yes	1	3	No
focmec	Yes	Yes	Yes	Yes	2	2	No
SOD	Yes	Yes	Yes	Yes	1	0	No
FLEXWIN	Yes	Yes	Yes	No	2	2	No
PRESTo	Yes	Yes	Yes	Yes	1	0	No
Earthworm	Yes	Yes	Yes	Yes	3	0	No
Mineos	Yes	Yes	Yes	No	1	0	No
SAC	Yes	Yes	Yes	No	1	3	No

Table 8 Reliability, Installation Break: The software “break” during installation. Tutorial Break: The software “break” during the initial tutorial testing.

Name	Installation Break	Tutorial Break
rfsyn	Yes	N/A
PITSA	Yes	N/A
RECFUNK09	No	N/A
Jpitsa	No	N/A
Station Analysis Tools	Yes	N/A
fissuresutil	N/A	N/A
OregonDSP	Yes	N/A
evalresp	No	N/A
jAmaSeis	No	N/A
SEISMIC_CPML	No	N/A
Seismic Handler	No	N/A
AIMBAT	No	Yes
JRG	No	Yes
NonLinLoc	No	Yes
tracedsp	No	N/A
CWP/SU	No	NA
GEE	No	No
TauP	No	No
msmod	No	N/A
FilterPicker	No	No
FreeUSP	No	Yes
iaspei-tau	No	N/A
JEvalResp	No	No
focmec	No	No
SOD	No	No
FLEXWIN	No	No
PRESTo	No	No
Earthworm	Yes	N/A
Mineos	No	No
SAC	No	No

Table 9 Usability, Tut: Getting Started Tutorial, Ex: Standard Example, UM: User Manual, Look/Feel: Usual look and feel of software, Vis: Lack of visibility (Norman's Principle) , User Char: User characteristics documented.

Name	Tut	Ex	UM	Look/feel	Vis	User Char	User support
rfsyn	No	No	Yes	Yes	No	Yes	Email
PITSA	No	No	No	Yes	No	N/A	No
RECFUNK09	Yes	No	Yes	Yes	No	N/A	E-mail
Jpitsa	No	No	No	Yes	No	Yes	Email
Station Analysis Tools	No	No	Yes	Yes	No	N/A	E-mail/Community
fissuresutil	No	No	No	N/A	No	N/A	No
OregonDSP	No	No	Yes	N/A	No	N/A	E-mail
evalresp	Yes	Yes	Yes	Yes	No	No	Community
jAmaSeis	No	No	Yes	Yes	Yes	Yes	E-mail
SEISMIC_CPML	No	No	Yes	Yes	No	No	Community/Email
Seismic Handler	No	No	Yes	Yes	No	N/A	Email/Community
AIMBAT	Yes	Yes	Yes	Yes	No	No	Email
JRG	Yes	No	Yes	Yes	No	No	Email
NonLinLoc	Yes	No	Yes	Yes	No	Yes	Email
tracedsp	No	Yes	Yes	Yes	No	No	E-mail
CWP/SU	Yes	Yes	Yes	Yes	No	No	E-mail/Seminar
GEE	Yes	Yes	Yes	Yes	Yes	No	No
TauP	Yes	Yes	Yes	N/A	No	No	E-mail
msmod	No	Yes	Yes	Yes	No	No	No
FilterPicker	Yes	Yes	Yes	Yes	Yes	N/A	Email
FreeUSP	Yes	Yes	Yes	Yes	No	N/A	E-mail/Community
iaspei-tau	Yes	Yes	Yes	Yes	No	No	E-mail
JEvalResp	Yes	Yes	Yes	Yes	No	No	E-mail
focmec	Yes	Yes	Yes	Yes	No	No	Email
SOD	Yes	Yes	Yes	Yes	No	No	E-mail
FLEXWIN	Yes	Yes	Yes	Yes	No	No	Community
PRESTo	Yes	Yes	Yes	Yes	No	No	E-mail/Community
Earthworm	Yes	Yes	Yes	Yes	No	N/A	E-mail/Community
Mineos	Yes	Yes	Yes	Yes	No	No	Community/Email
SAC	Yes	Yes	Yes	Yes	Yes	No	E-mail/Community

Table 10 Correctness and Verifiability, Library: Use of standard libraries, SRS: Software Requirements Specification, Evidence: Evidence to build confidence? Example: Standard Example explained?

Name	Library	SRS	Evidence?	Example
rfsyn	No	No	No	N/A
PITSA	No	No	No	No
RECFUNK09	Yes	No	No	No
Jpitsa	Yes	No	No	No
Station Analysis Tools	No	No	No	Yes
fissuresutil	Yes	No	No	No
OregonDSP	No	No	Yes	No
evalresp	No	No	No	Yes
jAmaSeis	Yes	Yes	N/A	N/A
SEISMIC_CPML	No	Yes	No	No
Seismic Handler	No	No	No	No
AIMBAT	No	No	Yes	No
JRG	No	No	Yes	No
NonLinLoc	Yes	No	Yes	Yes
tracedsp	Yes	No	No	No
CWP/SU	Yes	No	No	Yes
GEE	Yes	No	No	Yes
TauP	Yes	No	No	Yes
msmod	No	Yes	Yes	No
FilterPicker	No	No	Yes	No
FreeUSP	Yes	Yes	Yes	No
iaspei-tau	No	No	Yes	Yes
JEvalResp	Yes	No	No	Yes
focmec	Yes	Yes	Yes	Yes
SOD	Yes	No	Yes	Yes
FLEXWIN	Yes	No	Yes	Yes
PRESTo	No	Yes	Yes	N/A
Earthworm	Yes	No	Yes	N/A
Mineos	No	No	Yes	Yes
SAC	Yes	No	Yes	Yes

Table 11 Maintainability, VH: Versions history available, RC: Information on reviewing and contributing , log: Change log available, MT: Maintenance Type, Issue: Issue Tracking Tool, Bugs: Majority of Bugs fixed , VS: Versioning system used, Evidence: Any evidence that maintainability was considered in design, Cor: Corrective, Pfc: Perfective, Adp: Adaptive, Clone: Are there code clones?

Name	VH	RC	log	MT	Issue	Bugs	VS	Evidence	Clone
rfsyn	No	No	Yes	Cor	No	Yes	No	No	No
PITSA	No	No	Yes	Cor	No	Yes	No	Yes	No
RECFUNK09	No	No	No	Unclear	No	?	No	Yes	No
Jpitsa	No	No	No	Unclear	No	?	No	No	N/A
Station Analysis Tools	No	No	Yes	Cor	?	Yes	SVN	No	?
fissuresutil	No	No	No	N/A	?	?	Git	Yes	?
OregonDSP	Yes	No	No	Unclear	?	?	?	Yes	No
evalresp	No	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	No
jAmaSeis	No	N/A	No	?	Yes	Yes	?	Yes	?
SEISMIC_CPML	Yes	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	Git	Yes	No
Seismic Handler	Yes	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	Git/SVN	Yes	No
AIMBAT	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	Git	Yes	No
JRG	No	No	Yes	Adp/Pfc	?	Yes	?	Yes	No
NonLinLoc	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	No
tracedsp	Yes	No	Yes	Cor/Pfc	Yes	Yes	SeisCode	Yes	No
CWP/SU	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	?
GEE	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	?
TauP	Yes	No	No	Cor/Adp/Pfc	?	Yes	?	Yes	?
msmod	No	No	Yes	Adp/Pfc	?	?	SVN	Yes	No
FilterPicker	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	SVN	Yes	No
FreeUSP	Yes	Yes	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	No
iaspei-tau	No	No	Yes	Pfc	No	Yes	No	Yes	No
JEvalResp	No	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	?
focmec	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	No
SOD	Yes	No	Yes	Cor	?	Yes	?	Yes	?
FLEXWIN	Yes	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	Git	Yes	?
PRESTo	Yes	No	Yes	Cor/Adp/Pfc	?	Yes	?	Yes	No
Earthworm	Yes	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	SVN	Yes	No
Mineos	Yes	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	Git	Yes	No
SAC	No	Yes	Yes	Cor/Adp/Pfc	Yes	Yes	?	Yes	No

Table 12 Understandability (of code), Format: Consistent Identation and formatting, Coding st: Explicit coding standard, Id: Distinctive, Meaningful identifiers name, Constants: Constants (other than 0 or 1) hard coded, PC: proper comments, Algo: Reference to algorithm used, Mod: Code is modularised, Para: Parameters are in same order, SC: Descriptive names of source code files, DD: Design document present.

Name	Format	Coding st	Id	Constants	PC	Algo	Mod	para	SC	DD
rfsyn	Yes	No	No	Yes	Yes	No	Yes	N/A	Yes	No
PITSA	Yes	No	No	No	No	No	Yes	Yes	Yes	No
RECFUNK09	Yes	No	Yes	Yes	Yes	Yes	Yes	N/A	Yes	No
Jpitsa	N/A	No	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Station Analysis Tools	Yes	No	No	Yes	Yes	Yes	Yes	No	Yes	No
fissuresutil	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
OregonDSP	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No
evalresp	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	No
jAmaSeis	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SEISMIC_CPML	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
Seismic Handler	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	No
AIMBAT	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
JRG	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No
NonLinLoc	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
tracedsp	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No
CWP/SU	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
GEE	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	No
TauP	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No
msmod	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No
FilterPicker	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
FreeUSP	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
iaspei-tau	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
JEvalResp	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes	No
focmec	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
SOD	Yes	Yes	Yes	No	Yes	No	Yes	Yes	Yes	No
FLEXWIN	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No
PRESTo	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
Earthworm	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Mineos	Yes	No	Yes	No	Yes	Yes	Yes	Yes	Yes	No
SAC	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes	No

Table 13 Robustness, Performance and Reusability, install: Software break during installation , test: Software break during initial tutorial testing, Wrong I/P handling: Handling of wrong input by software , format: Can the software gracefully handle a change of the format of text input files where the end of line follows a different convention? Perf: Evidence that performance was considered? S/W reused: any portions of the software used by another package? Reuse Evidc: Any evidence of reusability?

Name	Wrong I/P handling	Format	Perf	S/W reused	Reuse Evidc
rfsyn	?	?	No	?	No
PITSA	?	?	No	No	No
RECFUNK09	?	?	No	?	No
Jpitsa	Yes	Yes	No	No	No
Station Analysis Tools	?	?	Yes	No	No
fissuresutil	?	?	No	Yes	Yes
OregonDSP	?	?	No	Yes	Yes
evalresp	?	?	Yes	Yes	?
jAmaSeis	N/A	N/A	Yes	No	No
SEISMIC_CPML	?	?	No	?	No
Seismic Handler	?	No	No	Yes	Yes
AIMBAT	?	?	Yes	No	No
JRG	Yes	Yes	No	Yes	Yes
NonLinLoc	Yes	Yes	No	No	?
tracedsp	Yes	Yes	Yes	?	N/A
CWP/SU	N/A	N/A	Yes	?	No
GEE	N/A	N/A	Yes	No	No
TauP	Yes	Yes	No	Yes	Yes
msmod	Yes	N/A	Yes	Yes	Yes
FilterPicker	No	N/A	No	Yes	Yes
FreeUSP	?	?	Yes	Yes	Yes
iaspei-tau	Yes	Yes	No	No	No
JEvalResp	?	Yes	Yes	Yes	Yes
focmec	Yes	Yes	No	No	No
SOD	N/A	N/A	No	Yes	Yes
FLEXWIN	Yes	YES	Yes	?	Yes
PRESTo	N/A	N/A	Yes	N/A	N/A
Earthworm	?	?	Yes	Yes	Yes
Mineos	Yes	Yes	Yes	Yes	No
SAC	Yes	Yes	Yes	Yes	Yes

Table 14 Portability, Platforms: Platforms specified for the software to work on , Port in code: How portability is handled (If source code given), All: Linux, Windows and Mac OS, Port not imp: Portability explicitly identified as not important, Evid in doc.: Convincing evidence present in documentation for portability?

Name	Platforms	Port in code	Port not imp	Evid in doc
rfsyn	Linux/Solaris	N/A	N/A	No
PITSA	Linux	N/A	N/A	No
RECFUNK09	Mac	N/A	Yes	No
Jpitsa	Linux/Win	Yes	Implicity	No
Station Analysis Tools	Linux	No	Implicity	No
fissuresutil	Unspecified	Yes	Implicitly	No
OregonDSP	Unspecified	N/A	Implicity	No
evalresp	Linux/Win	Yes	Implicity	No
jAmaSeis	All	N/A	No	Yes
SEISMIC_CPML	All	N/A	Implicity	No
Seismic Handler	Linux	Yes	Implicity	No
AIMBAT	All	Yes	No	Yes
JRG	All	Yes	No	Yes
NonLinLoc	Linux,Mac	Yes	No	No
tracedsp	Linux/Win32	Yes	No	Yes
CWP/SU	Linux	N/A	Implicitly	No
GEE	All	Yes	No	Yes
TauP	All	Yes	Implicitly	No
msmod	Win/Linux	Yes	No	Yes
FilterPicker	All	No	Implicity	No
FreeUSP	Linux	N/A	Implicity	N/A
iaspei-tau	Mac/Linux	Yes	No	Yes
JEvalResp	All	Yes	No	Yes
focmec	Solaris/Linux/Mac	Yes	Implicitly	No
SOD	Unix/Win	Yes	Implicitly	No
FLEXWIN	Linux	N/A	Implicity	No
PRESTo	ALL	Yes	No	Yes
Earthworm	All	Yes	No	Yes
Mineos	All	Yes	No	Yes
SAC	All	Yes	No	Yes

Table 15 Interoperability, External package: Software communicates with external package, Workflow uses other s/w: Workflow uses other software, API: External interactions (API) defined?

Name	External package	workflow uses other s/w	API
rfsyn	Yes	No	N/A
PITSA	Yes	No	N/A
RECFUNK09	Yes	No	N/A
Jpitsa	Yes	No	No
Station Analysis Tools	Yes	No	N/A
fissuresutil	Yes	No	Yes
OregonDSP	Yes	No	Yes
evalresp	Yes	No	No
jAmaSeis	Yes	Yes	N/A
SEISMIC_CPML	Yes	No	No
Seismic Handler	Yes	No	Yes
AIMBAT	Yes	No	N/A
JRG	Yes	No	Yes
NonLinLoc	Yes	No	No
tracedsp	Yes	Yes	No
CWP/SU	Yes	No	Yes
GEE	Yes	No	N/A
TauP	Yes	No	Yes
msmod	Yes	No	Yes
FilterPicker	Yes	No	Yes
FreeUSP	Yes	Yes	Yes
iaspei-tau	Yes	Yes	No
JEvalResp	Yes	No	Yes
focmec	Yes	No	No
SOD	Yes	No	Yes
FLEXWIN	Yes	No	N/A
PRESTo	Yes	N/A	N/A
Earthworm	Yes	Yes	Yes
Mineos	Yes	No	Yes
SAC	Yes	No	Yes

Table 16 Visibility/Transparency and Reproducibility, Dev Pro: Development process defined, Ease of Exam: Ease of examination relative to other software (out of 10), Dev/Env Rec: Record of development environment, Test Data: Availability of test data for verification, Auto Rep tool: Automated tools used to capture experimental data.

Name	Dev Pro	Ease of Exam	Dev/Env Rec	Test Data	Auto Rep tool
rfsyn	No	1	No	No	No
PITSA	?	2	No	Yes	No
RECFUNK09	?	3	No	No	No
Jpitsa	No	2	No	Yes	No
Station Analysis Tools	?	7	No	No	No
fissuresutil	?	5	No	No	No
OregonDSP	?	3	No	No	No
evalresp	No	5	No	No	No
jAmaSeis	No	5	No	No	No
SEISMIC_CPML	No	8	No	No	No
Seismic Handler	?	5	No	No	No
AIMBAT	?	7	No	Yes	No
JRG	?	7	No	Yes	No
NonLinLoc	?	6	No	Yes	No
tracedsp	No	2	No	No	No
CWP/SU	?	8	No	Yes	No
GEE	?	7	No	No	No
TauP	?	8	Yes	Yes	No
msmod	?	10	No	Yes	No
FilterPicker	?	7	No	Yes	No
FreeUSP	Yes	6	Yes	Yes	No
iaspei-tau	No	9	No	Yes	No
JEvalResp	No	9	No	Yes	No
focmec	?	10	No	Yes	No
SOD	?	7	No	Yes	No
FLEXWIN	No	8	Yes	Yes	No
PRESTo	Yes	8	No	Yes	No
Earthworm	Yes	10	Yes	Yes	No
Mineos	No	8	No	No	No
SAC	?	8	No	Yes	?