Practical Software Engineering Strategies for Scientific Computing

Christopher J. Roy
Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061

Software engineering is a formal set of tools and procedures for the efficient development of reliable software. Scientific computing software differs from other types of software because the “correct” outputs for a given set of software inputs are generally not known. Since scientific computing relies on approximate numerical solutions to partial differential or integral equations, the answers produced by the software will depend on the mesh employed, the time step chosen, the iterative convergence tolerance used, the number of significant figures used in the floating point computations, etc. The goal of this paper is to provide a broad overview of modern software engineering practices, with a focus on tools and methods that are particularly relevant to scientific computing. Specific topics that are emphasized include software development models, choice of programming language, version control systems, static analysis tools, dynamic testing procedures, and the quantitative assessment of software quality and reliability.

I. Introduction

Software engineering encompasses the tools and methods for defining requirements for, designing, programming, testing, and managing software. It consists of monitoring and controlling both the software processes and the software products to ensure reliability. Software engineering was developed primarily from within the computer science community, and its use is essential for large software development projects and for high-assurance software systems such as those for aircraft control systems, nuclear power plants, and medical devices (e.g., pacemakers). Software engineering is an enormously broad subject which has been addressed by numerous books (e.g., Sommerville, 2004; Pressman, 2005; McConnel, 2005) as well as a broad array of content on the World Wide Web (e.g., SWEBOK, 2004; Eddins, 2006; Wilson, 2009).

Computational scientists and engineers generally receive no formal training in modern software engineering practices. Our own search of the software engineering literature found a large number of contributions in various textbooks, on the web, and in scientific articles – mostly dominated by software engineering practices and processes that do not consider some of the unique aspects of scientific software. The goal of this paper is to give a brief overview of software engineering practices with a particular focus on those practices that are of particular importance to scientific computing software. An expanded version of this paper can be found in Chapter 4 of the book by Oberkampf and Roy (2010).

II. Software Requirements

A software requirement is a “property that must be exhibited in order to solve some real-world problem” (SWEBOK, 2004). Uncertainty in requirements is a leading cause of failure in software projects (Post and Kendall, 2004). While it is certainly ideal to have all requirements rigorously specified at the beginning of a software project, this can be difficult to achieve for scientific software. Especially in the case of large scientific software development projects, complete requirements can be difficult to specify due to rapid changes in models, algorithms, and even in the specialized computer architectures used to run the software. While lack of requirements definition can adversely affect the development of scientific software, these negative effects can be mitigated somewhat if close communication is maintained between the developer of the software and the user (Post and Kendall, 2004).

There are two main types of software requirements. User requirements are formulated at a high level of abstraction, usually in general terms which are easily understood by the user. Software system requirements, on the

1 Associate Professor, Aerospace and Ocean Engineering Department, Associate Fellow, AIAA.
other hand, are a precise and formal definition of a software system’s functions and constraints. The software system requirements are further decomposed as follows:

1) functional requirements – rigorous specifications of required outputs for a given set of inputs,
2) non-functional requirements – additional non-functional constraints such as programming standards, reliability, and computational speed, and
3) domain requirements – those requirements that come from the application domain such as a discussion of the partial differential or integral equations to be solved numerically for a given scientific computing application.

The domain requirements are crucial in scientific computing since these will be used to define the specific governing equations, models, and numerical algorithms to be implemented. Finally, if the software is to be integrated with existing software, then additional specifications may be needed for the procedure interfaces (application programming interfaces, or APIs), data structures, or data representation (e.g., bit ordering) (Sommerville, 2004).

III. Software Development

A. Software Process Models and Architectural Design

Software development encompasses the design, construction, and maintenance of software. While software testing should also be an integral part of the software development process, we will defer a detailed discussion of software testing until the next section. There are three main software process models. In the traditional waterfall model, the various aspects of the software development process (requirements specification, architectural design, programming, testing, etc.) are decomposed into separate phases, with each phase beginning only after the previous phase is completed. In response to criticisms of the waterfall software development model, a competing approach called iterative and incremental development was proposed. This iterative, or evolutionary, development model is based on the idea of interweaving each of the steps in the software development process, thus allowing customer input early in the development process through software prototypes which may initially have only limited capabilities. These software prototypes are then refined based on the customer input, resulting in software with increasing capability. A third model, component-based software engineering, can be used when a large number of reusable components are available, but has limited use in scientific computing. Most modern software development models, such as the rational unified process (Sommerville, 2004) and agile software development, are based on the iterative and incremental development model.

Software architectural design is the process of identifying software sub-systems and their interfaces before any programming is done (Sommerville, 2004). The primary product of architectural design is usually a flowchart (or series of flowcharts) which describes the software sub-systems and their structure. A software sub-system is defined as a subset of the full software system that does not interact with other sub-systems. Each software sub-system is made up of components, which are sub-sets of the full system which interact with other components. Components may be based on a procedural design (subroutines, functions, etc.) or an object-oriented design, and both approaches are discussed in more detail in the next section.

B. Programming Languages

There are a variety of factors to consider when choosing a programming language. The two main programming paradigms used in scientific computing are procedural programming and object-oriented programming. Procedural programming relies on calls to different procedures (routines, subroutines, methods, or functions) to execute a series of sequential steps in a given programming task. A significant advantage of procedural programming is that it is modular, i.e., it allows for reuse of procedures when tasks must be performed multiple times. In object-oriented programming, the program is decomposed into objects which interact with each other through the sending and receiving of messages. Objects typically make use of private data which can only be accessed through that object, thus providing a level of independence to the objects. This independence makes it easier to modify a given object without impacting other parts of the code. Object-oriented programming also allows for the reusability of components across the software system. Most modern, higher-level programming languages used for scientific computing support both procedural and object-oriented programming. Scientific programming languages that are primarily procedural include BASIC, C, Fortran, MATLAB, Pascal, and Perl, while those that are primarily object-oriented include C++, Java, and Python.
Low level computing languages such as machine language and assembly language (often found in simple electronic devices) execute extremely fast, but require additional time and effort during the programming and debugging phases. One factor to consider is that high-level languages, which often make use of more natural language syntax and varying levels of programming abstraction, have the advantage of making programming complex software projects easier, but will not execute as fast as a lower-level programming language. A qualitative comparison of selected programming languages is shown in Figure 1 which compares programming effort to execution speed. In scientific computing, the higher-level programming languages such as Python, MATLAB, and Java are ideal for small projects and prototyping, while production-level codes are usually programmed in C, C++, or Fortran due to the faster execution speeds.

![Figure 1. Qualitative example of programming time versus execution time (adapted from Wilson, 2009).](image)

Another factor to consider when choosing a programming language is the impact on the software defect rate and the subsequent maintenance costs. Here we define a software defect as an error in the software that could potentially lead to software failure and the software defect rate as the number of defects per 1000 lines of executable code. Evidence suggests that the software defect rate is at best weakly dependent on the choice of programming language (Hatton, 1997a). However, Hatton (1996) found that defects in object-oriented languages can be more expensive to find and fix, possibly by as much as a factor of three.

Standards for programming languages are generally developed by a costly and complex process. However, most programming language standards still contain coding constructs that are prone to producing software failures. These failure-prone constructs can arise in a number of different ways (Hatton, 1997a) including simple oversight by the standards process, lack of agreement on the content of the standards, because the decision was explicitly made to retain the functionality provided by the construct, or because of errors in the programming language standards documentation. In some cases, less fault-prone subsets of a programming language exist which reduce or eliminate the presence of the dangerous coding constructs. One example of a safe subset for the C programming language is Safer C (Hatton, 1995).

C. Refactoring

Oftentimes, at the end of a software development effort, the developer realizes that choices made early in the software design phase have led to computationally inefficient or cumbersome programming. Refactoring is the act of modifying software such that the internal software structure is changed, but the outward behavior is not. Refactoring can reduce the complexity, computational time, and/or memory requirements for scientific software. However, refactoring should not be undertaken until a comprehensive test suite (see Section V) is in place to ensure that the external behavior is not modified and that programming errors are not introduced.
IV. Configuration Management

Configuration management deals with the control and management of the software products during all phases of the software product’s lifecycle including planning, development, production, maintenance, and retirement. Here software products include not only the source code, but also user and theory manuals, software tests, test results, design documents, web pages, and any other items produced during the software development process. Configuration management tracks the way software is configured over time and is used for controlling changes to, and for maintaining integrity and traceability of, the software products (Sommerville, 2004). The key aspects of configuration management include using version (or revision) control for source code and other important software products, identification of the software products to be managed, recording, approving, and tracking issues with the software, managing software releases, and ensuring backups are made. This section will focus on the first of these: software version control.

A. Version Control

Version control tracks changes to source code or other software products. A good version control system can tell you what was changed, who made the change, and when the change was made. It allows a software developer to undo any changes to the code, going back to any prior version. This can be particularly helpful when a researcher/developer is trying to reproduce results from an earlier paper/report and merely requires documentation of the version number or the date the results were generated. Version control also provides a mechanism for incorporating changes from multiple developers, an essential feature for large software projects or projects with geographically remote developers. All source code should be maintained in a version control system, regardless of how large or small the software project (Eddins, 2006). Some key concepts pertaining to version control are discussed below (Collins-Sussman et al., 2009). Note that the generic descriptor “file” is used which could represent not only source code and other software products, but also any other type of file stored on a computer.

- **Repository** – single location where the current and all prior versions of the files are stored
- **Working copy** – the local copy of a file from the repository which can be modified and then checked in or “committed” to the repository
- **Check-out** – the process of creating a working copy from the repository (either the current version or an earlier version)
- **Check-in** – a check-in or commit occurs when changes made to a working copy are merged into the repository
- **Diff** – a summary of the differences between a working copy and a file in the repository, often taking the form of the two files side-by-side with differences highlighted
- **Conflict** – a conflict occurs when two or more developers attempt to make changes to the same file and the system is unable to reconcile the changes (note: conflicts generally must be resolved by either choosing one version over the other or by integrating the changes from both into the repository by hand)
- **Update** – merges recent changes to the repository into a working copy

The basic steps that one would use to get started with a version control tool are as follows. First, a repository is created, ideally on network server which is backed up frequently. Then a project (directory structure and/or files) is imported to the repository. This initial version can then be checked-out as a working copy. The project can then be modified in the working copy, with the differences between the edited working copy and the original repository version examined using a diff procedure. Finally, the working copy of the project can be checked-in to the repository, generating a new version of the project.

B. Recommended Version Control Systems

There is a wide array of version control systems available to the software developer. These systems range from free, open-source systems such as CVS, OpenCVS, and Subversion to commercially available systems such as Accurev, Perforce, and Visual SourceSafe. A detailed list can be found at: en.wikipedia.org/wiki/List_of_revision_control_software. While the most well-known, freely-available version control system is CVS (Concurrent Versions Systems), a popular and more modern alternative is Subversion (SVN), which is generally seen as the successor to CVS. Subversion is also freely-available (at subversion.tigris.org), is open-source and actively maintained, and has a very useful, free book available online (Collins-Sussman et al., 2008). Subversion has a number of easy-to-use Graphical User Interfaces for all major computing platforms. Eddins (2006) recommends using TortoiseSVN (tortoisesvn.tigris.org) on Windows platforms (which integrates into
Windows Explorer) and RapidSVN (rapidsvn.tigris.org) on Linux and Macintosh platforms. A short tutorial showing the basic steps for using TortoiseSVN for version control on a Windows system can be found at: www.aoe.vt.edu/~cjroy/MISC/TortoiseSVN-Tutorial.pdf.

One of the most useful features of most version control software is the ability to rapidly analyze changes in source code between different versions. A screen shot from the TortoiseSVN diff tool is presented in Figure 2. In this example, the current working copy (shown on the right) is being compared to version 7 from the repository. Lines that have any differences between the two versions of the code are highlighted in light blue, while specific differences in variable values, etc. are shown in red in the original version and in yellow in the working copy. Lines deleted from the original version are highlighted in orange, while lines added to the working copy are highlighted in yellow. In addition, by simply placing the cursor over any line in either version, the corresponding lines from both versions are compared vertically at the bottom of the window, making the visual inspection of differences even easier.

Figure 2. Example of the TortoiseSVN diff tool for showing difference between file versions.

V. Software Verification and Validation

The definitions accepted by AIAA (1998) and ASME (2006) for verification and validation as applied to scientific computing address the mathematical accuracy of a numerical solution (verification) and the physical accuracy of a given model (validation); however, the definitions used by the software engineering community (e.g., ISO, 1991; IEEE, 1991) are different. In software engineering, verification is defined as ensuring that software conforms to its specifications (i.e., requirements) and validation is defined as ensuring that software meets the customer’s expectations. Some argue that these definitions are really the same; however, upon closer examination, they are in fact different.

The key differences in these definitions for verification and validation are due to the fact that in scientific computing, we begin with a governing partial differential or integral equation, which we will refer to as our model. For problems that we are interested in solving, there is generally no known exact solution to this model. It is for this reason that we must develop numerical approximations to the model (i.e., the numerical algorithm) and then implement that numerical algorithm within scientific computing software. Thus the two striking differences between how the scientific computing community and the software engineering community define verification and validation are as follows. First, in scientific computing, validation requires a comparison to experimental data. Furthermore, one can attempt to perform validation assessments on models, but not on the software itself (e.g., what if “validated”
software was used to solve a problem on an improper grid or with insufficient iterative convergence?). The software engineering community defines validation of the software as meeting the customer’s expectations, which is, in our opinion, too vague to tie it back to the definition from scientific computing. Second, in scientific computing, there is generally no true system level software test for real problems of interest. The “correct” output from the scientific software must depend on the number of significant figures used in the computation, the computational mesh resolution and quality, the time step (for unsteady problems), the level of iterative convergence, etc. See Oberkampf and Roy (2010) for a discussion of system-level tests for scientific software as well as verification approaches for formally connecting the results produced by scientific software back to the models employed.

In this section, we will distinguish between the two definitions of verification and validation by inserting the word “software” when referring to the definitions from software engineering. Three additional definitions that will be used throughout this section are those for software defects, faults, and failures (Hatton, 1997b). A software defect is a coding mistake (bug) or the misuse of a coding construct that could potentially lead to a software failure. A software fault is a defect which can be detected without running the code, i.e., through static analysis. Examples of defects that can lead to software faults include dependence on uninitialized variables, mismatches in parameter arguments, and unassigned pointers. A software failure occurs when the software returns an incorrect result or when it terminates prematurely due to a run-time error (overflow, underflow, division by zero, etc.). Some examples of catastrophic software failures are given by Hatton (1997a).

A. Static Analysis

Static analysis is any type of assessment of software correctness that does not require program execution. Examples of static analysis methods include code inspection, peer review, compiling of the code, and the use of automatic static analyzers. Hatton (1997a) estimates that approximately 40% of software failures are due to static faults. Some examples of static faults are:
- dependency on uninitialized or undeclared variables,
- interface faults: too few, too many, or wrong type of arguments passed to a function/subroutine,
- casting a pointer to a narrow integral type (C), and
- use of non-local variables in functions/subroutines (Fortran).

All of these static faults, as well as others that have their origins in ambiguities in the programming language standards, can be prevented by using static analysis.

Any time the code is compiled it goes through some level of static analysis. The level of rigor of the static analysis often depends on the options used during compilation, but there is a trade-off between the level of static analysis performed by the compiler and the execution speed. Many modern compilers provide different modes of compilation such as a release mode, a debug mode, and a check mode that perform increasing levels of static analysis. Due to differences in compilers and operating systems, many software developers make it standard practice to compile the source code with different compilers and on different platforms.

Automatic static analyzers are external tools that are meant to complement the checking of the code by the compiler. They are designed to find inconsistent or undefined use of a programming language that the compiler will likely overlook, as well as coding constructs that are generally considered as unsafe. Some static analyzers available for C/C++ include the Safer C Toolset, CodeWizard, CMT++, Cleanscape LintPlus, PC-lint/FlexeLint, and QA C. Static analyzers for Fortran include floppy/fflow and ftnchek. There is also a recently-developed static analyzer for MATLAB called M-Lint (MATLAB, 2008). For a more complete list, or for references to each of these static analyzers, see www.testingfaqs.org/t-static.html.

B. Dynamic Testing

Dynamic software testing can be defined as the “dynamic verification of the behavior of a program on a finite set of test cases … against the expected behavior” (SWEBOK, 2004). Dynamic testing includes any type of testing activity which involves running the code. The types of dynamic testing discussed in this section include unit testing, component testing, system testing, and regression testing. The use of a test harness for performing dynamic testing in an automated fashion and the coverage of coding options with dynamic tests are also discussed in this section.

1. Unit Testing

Unit testing is used to verify the execution of a single routine (e.g., function, subroutine, object class) of the code (Eddins, 2006). Unit tests are designed to check for the correctness of routine output based on a given input. They should also be easy to write and run, and should execute quickly. Properly designed unit tests also provide examples
of proper routine use such as how the routine should be called, what type of inputs should be provided, what type of outputs can be expected.

While it does take additional time to develop unit tests, this extra time in code development generally pays off later in reduced time debugging. The author’s experience with even small scientific computing code development in university settings suggests that the typical ratio of debugging time to programming time for students is at least five to one. The wider the unit testing coverage (i.e., percentage of routines that have unit tests), the more reliable the code is likely to be. In fact, some software development strategies such as Extreme Programming require tests to be written before the actual routine to be tested is created. Such strategies require the programmer to clearly define the interfaces (inputs and outputs) of the routine up front.

2. Component Testing

Kleb and Wood (2006) make an appeal to the scientific computing community to implement the scientific method in the development of scientific software. Recall that in the scientific method, a theory must be supported with a corresponding experiment that tests the theory, and must be described in enough detail that the experiment can be reproduced by independent sources. They recommend testing at the component level, where a component is considered to be a sub-model or algorithm. Furthermore, they strongly suggest that model and algorithm developers publish test fixtures with any newly proposed model or algorithm. These test fixtures are designed to clearly define the proper usage of the component, give examples of proper usage, and give sample inputs along with correct outputs that can be used for testing the implementation into an independent code. An example of such a test fixture for Sutherland’s viscosity law is presented in Figure 3.

![Figure 3. Example of a component-level test fixture for Sutherland’s viscosity law (reproduced from Kleb and Wood, 2006).](image)

Component-level testing can be performed when the sub-model or algorithm are algebraic since the expected (i.e., correct) solution can be computed directly. However, for cases where the sub-model involves numerical approximations (e.g., many models for fluid turbulence involve differential equations), then the expected solution will necessarily be a function of the chosen discretization parameters, and the more sophisticated code verification methods discussed in Oberkampf and Roy (2010) should be used. For models that are difficult to test at the system level (e.g., the min and max functions significantly complicate the code verification process), the component-level testing for the models (or different parts of the model) can be used. Finally, if all components have been successfully tested individually, one should not get a false sense of security about how the software will behave at the system level. Complex interactions between components can only be tested at the system level.

3. System Testing

System-level software testing is testing of the entire code as a whole. For a given set of inputs to the code, what is the correct code output. In software engineering, system level testing is the primary means by which one determines if the software requirements have been met (i.e., software verification). For non-scientific software, it is often possible to a priori determine what the correct output of the code should be. However, for scientific computing software where partial differential or integral equations are solved, the “correct” output is generally not known.
ahead of time. Furthermore, the code output will depend on the grid and time step chosen, the iterative convergence level, the machine precision, etc. For scientific software, system-level testing is generally addressed through order of accuracy verification (e.g., see Oberkampf and Roy, 2010).

4. Regression Testing

Regression testing involves the comparison of code or software routine output to the output from earlier versions of the code. Regression tests are designed to prevent the introduction of coding mistakes by detecting unintended consequences of changes in the code. Regression tests can be implemented at the unit, component, or system level. In fact, all of the dynamic tests described above can also be implemented as regression tests. The main difference between regression testing and other dynamic testing is that regression tests do not compare code output to the correct expected value, but instead to the output from previous versions of the code. Careful regression testing combined with the other dynamic testing approaches can minimize the chances of introducing software defects during code development and maintenance.

5. Test Harness

Many different types of dynamic software tests have been discussed in this section. For larger software development projects, it would be extremely tedious if the developer had to run each of the tests separately and then examine the results. Especially in the case of larger development efforts, automation of software testing is a must. A test harness is the combination of software and test data used to test the correctness of a program or component by automatically running it under various conditions (Eddins, 2006). A test harness is usually composed of a test manager, test input data, test output data, a file comparator, and an automatic report generator. While it is certainly possible to create your own test harness, there are a variety of test harnesses that have been developed for a wide range of programming languages. For a detailed list, see: en.wikipedia.org/wiki/List_of_unit_testing_frameworks.

Once a suite of tests has been set up to run within a test harness, it is recommended that these tests be run automatically at specified intervals. Shorter tests can be run in a nightly test suite, while larger tests which require more computer time and memory may be set up in weekly or monthly test suites. In addition, an approach called continuous integration testing (Duvall et al., 2007) requires that specified test suites be run after every code check in, and that any code modifications be accompanied by new, or perhaps modified, unit tests (Sommerville, 2004).

6. Code Coverage

Regardless of how software testing is done, one important aspect is the coverage of the tests. Code coverage can be defined as the percentage of code components (and possibly their interactions) for which tests exist. While testing at the unit and component levels is relatively straightforward, system-level testing must also address interactions between different components. Large, complex scientific computing codes generally have a very large number of choices for models, submodels, numerical algorithms, boundary conditions, etc. Assume for the moment that there are 100 different options in the code to be tested, a conservative estimate for most production-level scientific computing codes. Testing each option independently (although generally not possible) would require 100 different system-level tests. Testing pair-wise combinations for interactions between these different options would require 4,950 system level tests for complete code coverage. Testing the interactions between groups of three would require 161,700 tests. While this is clearly an upper bound since certain interactions will be impossible, it does provide a sense of the magnitude of the task of achieving complete code coverage of model/algorithm interactions.

VI. Software Quality and Reliability

Recall that earlier software quality was defined as conformance to customer requirements and expectations. However, this definition of quality can really only be applied after the complete software product is delivered to the customer. Another aspect of software quality that we will find useful is software reliability. One definition of software reliability is the probability of failure-free operation of software in a given environment for a specified time (Musa, 1999). This section presents two quantitative approaches for measuring code quality and reliability. Defect density analysis provides an explicit measure of reliability, while complexity analysis provides an implicit measure of reliability. See the appendices for a discussion of recommended programming practices as well error-prone coding constructs that should be avoided when possible.
A. Defect Density Analysis

The most direct method for assessing the reliability of software is in terms of the number of defects in the software. Defects can lead to static errors (faults) and dynamic errors (failures). The defect density is usually reported as the number of defects per executable source lines of code (SLOC). Hatton (1997a) argues that it is only by measuring the defect density of software, through both static and dynamic testing, that an objective assessment of software reliability be made. Hatton’s T Experiments (Hatton, 1997b) are discussed in detail in Section VII and represent the largest known defect density study of scientific software.

B. Complexity Analysis

Complexity analysis is an indirect way of measuring code quality because it requires a model to convert certain internal quality attributes to code reliability (Sommerville, 2004). One possible model is to assume that a high degree of complexity in a component (function, subroutine, object class, etc.) is bad while a low degree of complexity is good. However, Hatton (1997a) used quantitative testing to show that in fact the defect density in components follows a U-shaped curve, with the minimum occurring at 150-250 lines of source code per component, independent of both programming language and application area. He surmised that the increase in defect density for smaller components may be related to the inadvertent adverse effects of component re-use (see Hatton (1996) for more details). Some different internal code attributes that can be used to indirectly assess code quality are discussed in this sub-section. In most cases, tools exist for evaluating these complexity metrics automatically.

Source Lines of Code (SLOC)

The simplest measure of complexity can be found by counting the number of executable source lines of code (SLOC) for each component. Hatton (1997a) recommends keeping components between 150 and 250 SLOC.

NPATH Metric

The NPATH metric simply counts the number of possible execution paths through a component (Nejmeh, 1988). Nejmeh (1988) recommends keeping this value below 200.

Cyclomatic Complexity

The cyclomatic, or McCabe, complexity (McCabe, 1976) is defined as one plus the number of decision points in a component, where a decision point is defined as any loop or logical statement (if, elseif, while, repeat, do, for, or, etc.). The maximum recommended value for cyclomatic complexity of a component is ten (Eddins, 2006).

VII. Case Study in Scientific Computing: The T Experiments

In the early 1990’s, Les Hatton undertook a broad study of scientific software reliability known collectively as the “T Experiments” (Hatton, 1997b). This study was broken into two parts: the first (T1) examined codes from a wide range of scientific disciplines using static analysis, while the second (T2) examined codes in a single discipline using dynamic testing.

The T1 study used static deep-flow analyzers to examine more than 100 different codes in 40 different application areas. All codes were written in C, FORTRAN 66, or FORTRAN 77, and the static analyzers used were QA C (for the C codes) and QA Fortran (for the FORTRAN codes). The main conclusion of the T1 study was that the C codes contained approximately 8 serious static faults per 1000 lines of executable code, while the FORTRAN codes contained approximately 12 faults per 1000 lines. A serious static fault is defined as a statically-detectable defect that is likely to cause the software to fail. For more details on the T1 study, see Hatton (1995).

The T2 study examined a subset of the codes from the T1 study in the area of seismic data processing which is used in the field of oil and gas exploration. This study examined nine independent, mature, commercial codes which employed the same algorithms, the same programming language (Fortran), the same user-defined parameters, and the same input data. Hatton refers to such a study as N-version programming since each code was developed independently by a different company. Each of the codes consisted of approximately 30 sequential steps, 14 of which used unambiguously defined algorithms, referred to in the study as primary calibration points. Agreement between the codes after the first primary calibration point was within 0.001% (i.e., within machine precision for single-precision computations); however, agreement after primary calibration point 14 was only within 100% (i.e., a factor of two). It is interesting to note that distribution of results from the various codes was found to be non-Gaussian with distinct groups and outliers, suggesting that the output from an N-version programming test should not be analyzed with Bayesian statistics. Hatton concluded that the disagreements between the different codes are
due primarily to software errors. Such dismal results from the T2 study prompted Hatton to conclude that “the results of scientific calculations carried out by many software packages should be treated with the same measure of disbelief researchers have traditionally attached to the results of unconfirmed physical experiments.” For more details on the T2 study, see Hatton and Roberts (1994).

These alarming results from Hatton’s “T Experiments” highlight the need for employing good software engineering practices in scientific computing. At a minimum, the simple techniques presented in this paper such as version control, static analysis, dynamic testing, and quality metrics should be employed for all scientific computing software projects to improve quality and reliability.

Appendix A: Recommended Programming Practices

The following is a list of recommended programming practices designed to increase the reliability of scientific computing software along with a brief description of each practice.

1. Use Strongly-Typed Programming Languages
   Although there is some ambiguity in the definition, here we refer to a strongly-typed programming language as one which 1) requires that a variable or object maintain the same type (e.g., integer, floating point number, character) during program execution and 2) has strict rules as to which types can be used during operations (i.e., implicit type conversions are not allowed). Common examples of the latter are the use of integer division on floating point numbers and the use real functions (e.g., \texttt{cos}, \texttt{log}) on integers. BASIC and C are considered weakly-typed languages, while C++, Java, Fortran, Pascal, and Python are considered strongly-typed. For type information on other programming languages, see [en.wikipedia.org/wiki/Template:Type_system_cross_reference_list](en.wikipedia.org/wiki/Template:Type_system_cross_reference_list). A type-safe program can be written in a weakly-typed language by using explicit type conversions (to convert integers to real numbers, real numbers to integers, etc.).

2. Use Safe Programming Language Subsets
   In order to avoid error-prone coding constructs, safe programming language subsets are recommended.

3. Use Static Analyzers
   Hatton (1997) estimates that approximately 40% of all software failures are due to static faults which are readily found with the use of static analyzers (see Section V).

4. Use Long, Descriptive Identifiers
   Most modern programming languages allow long, descriptive names to be used for variables, objects, functions, subroutines, etc. The extra time spent typing in these longer names will be more than made up by the reduction in time spent figuring out what a variable contains or what a routine does.

5. Write Self-Commenting Code
   The good software developer will endeavor to write code in such a way that the coding itself clearly explains its purpose (Eddins, 2006). This is certainly aided by the use of descriptive identifiers as discussed above. Code comments should still be used, but when 100 lines of comments are needed to explain the workings of 10 lines of executable source code, the source code is probably not well written.

6. Use Private Data
   Accidental over-writing of data can be minimized through the use of private data, where data is made available only to those objects and routines that need to process it. Both C++ and Fortran 90/95 allow for the use of private data, the latter through the use of \texttt{Public} and \texttt{Private} attributes within modules.

7. Use Exception Handling
   Exceptions can occur due to internal conditions (e.g., division by zero, overflow) or external factors (e.g., insufficient memory available, input file does not exist). Exception handling can be as simple as letting the user know the local state of the system and the location where the exception occurred, or it could transfer control to a separate exception-handling code. Some programming languages such as Java and C++ have built-in exception-handling constructs.
8. Use Indentation for Readability

Indent the coding blocks to denote different levels of looping structures and logical constructs to make them more readable. An example of an indented Fortran 95 code construct is given below.

```fortran
if(mms == 0) then
    !Set standard boundary conditions
    Call Set_Boundary_Conditions
elseif(mms == 1) then
    !Calculate exact solution for temperature
    do j = 1, jmax
        do i = 1, imax
            Temperature_MMS(i,j) = MMS_Exact_Solution(x,y)
        enddo
    enddo
    !Set MMS boundary conditions
    Call Set_MMS_Boundary_Conditions
else
    !Check for invalid values of variable mms
    write(*,*) 'Error: mms must equal 0 or 1 !!! '
    Error_Flag = 1
endif
```

9. Use Module Procedures (Fortran only)

The use of module procedures (functions and subroutines) in Fortran rather than standard procedures provides an explicit interface between the procedure and its calling program. Thus interface consistency can be checked and interface errors can be found during code compilation rather than at run time.

**Appendix B: Error-Prone Programming Constructs**

Although allowed in certain programming languages, the following programming constructs are known to be error prone and should be avoided when possible. However, in some cases (e.g., dynamic memory allocation, pointers, parallelism) these programming constructs may be unavoidable.

1. **Implicit Type Definitions**

Implicit variable type definitions, where new variables can be introduced in a program without a corresponding type specification, should be avoided. For the Fortran programming language, this means that “Implicit None” should appear at the beginning of every component.

2. **Mixed-Mode Arithmetic**

With the exception of exponentiation, integer and real variables should not be used in a single expression. When they do occur together, explicit type conversions (e.g., real to integer or integer to real) should be used.

3. **Duplicate Code**

When the same coding construct appears multiple times in a program, it is a good indication that the piece of coding should be replaced with a function or subroutine. Duplicate code can make software development tedious since a modification to one instance requires the developer to also search out all other instances of the repeated code. Eddins (2006) cautions that “anything repeated in two or more places will eventually be wrong in at least one.”

4. **Equality Checks for Floating Point Numbers**

Since floating point (real) numbers are subject to machine round-off errors, equality comparisons between them should be avoided. For example, instead of checking for equality between the floating point numbers $A$ and $B$, one could instead check to see if the absolute value of $(A - B)$ is less than a specified tolerance.

5. **Dynamic Memory Allocation**

Dynamic memory allocation occurs when a program allocates memory at run-time rather than compile time. Memory allocation errors are difficult to detect and can result in unpredictable behavior over time, thus dynamic memory allocation should be avoided if possible.
6. **Recursion**

Recursion occurs when a component calls itself, either directly or indirectly. A recursive programming construct can be difficult to analyze, and errors in a recursive program can lead to the allocation of a system’s entire available memory (Sommerville, 2004).

7. **Pointers**

A pointer is a programming construct that contains the address of a direct location in machine memory (Sommerville, 2004). The use of pointers should be avoided since pointer errors can cause unexpected program behavior (e.g., see aliasing below) and can be extremely difficult to find and correct.

8. **Aliasing**

Aliasing “occurs when more than one name is used to refer to the same entity in a program” (Sommerville, 2004) and should be avoided.

9. **Inheritance**

Inheritance occurs when an object “inherits” some characteristics from another object. Objects that employ inheritance are more difficult to understand since their defining characteristics are located in multiple locations in the program.

10. **GOTO Statements**

GOTO statements should be avoided as they make the program difficult to follow, often resulting in a complex and tangled control structure (i.e., “spaghetti” code).

11. **Parallelism**

Although the use of parallel processing in large scientific computing applications is usually unavoidable, the developer should be aware of the potential for unexpected behavior due to timing interactions between processes. In general, these issues cannot be detected with static analysis and may be platform dependent.

**References**


