Using Code Mutation to Study Code Faults in Scientific Software

by

Daniel Alan Hook

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
April 2009

Copyright © Daniel Alan Hook, 2009
Abstract

Code faults can seriously degrade scientific software accuracy. Therefore, it is imperative that scientific software developers scrutinize their codes in an attempt to find these faults. This thesis explores, particularly, the efficacy of code testing as a method of scientific software code fault detection.

Software engineers, as experts in code quality, have developed many code testing techniques, but many of these techniques cannot readily be applied to scientific codes for at least two reasons. First, scientific software testers do not usually have access to an ideal oracle. Second, scientific software outputs, by nature, can only be judged for accuracy and not correctness. Testing techniques developed under the assumption that these two problems can be ignored—as most have been—are of questionable value to computational scientists.

To demonstrate the reality of these problems and to provide an example of how software engineers and scientists can begin to address them, this thesis discusses the development and application of a novel technique: Mutation Sensitivity Testing (MST). MST is based on traditional mutation testing, but—in place of a focus on mutant “killing”—MST focuses on assessing the mutation sensitivity of a test set.

In this thesis, MST experiments are conducted using eight small numerical routines, four classes of mutation operators, and 1155 tests. The results are discussed
and some conclusions are drawn. Two observations are of particular interest to computational scientists. First, it is found that oracles that exhibit uncertainties greater than (approximately) 80% of the expected output are of questionable value when they are used in the testing of scientific software. Second, it is found that a small number of carefully selected tests may be sufficient to falsify a code.
Acknowledgments

I would like to thank my supervisor Dr. Diane Kelly for her support and guidance. I also thank my wife Meredith for her support: she helped me find the time to complete this thesis. Additional thanks to my parents for their support and to my son Aeryck for letting me get some sleep when I needed it.
# Table of Contents

Abstract i

Acknowledgments iii

Table of Contents iv

List of Tables vii

List of Figures viii

Chapter 1:

**Introduction** ................................. 1

1.1 What is this Thesis About? .......................... 1

1.2 Why is this Work Relevant? .......................... 2

1.3 An Outline of the Work ............................. 5

Chapter 2:

**Important Concepts** ............................. 7

2.1 Accuracy and Error ............................... 8

2.2 Identifying Unacknowledged Errors .................. 12

2.3 Important Testing Concepts ........................ 19
List of Tables

5.1 Counts of categorized mutants. . . . . . . . . . . . . . . . . . . . . . 68
5.2 Counts of minimally equivalent tests and key detectors. . . . . . . . 69
List of Figures

2.1 Some errors that can accumulate as a result of domain transitions. . . 11
2.2 The roles of validation, verification, and scrutinization. . . . . . . . 18

5.1 Maximum observed error ($\Gamma$) distribution for $T_{cmb}$ (full domain). . . 70
5.2 Maximum observed error ($\Gamma$) distribution for $T_{cmb}$ (zoomed-in). . . 71
5.3 Detection scores accumulated from all targets (full domain). . . . . 72
5.4 Detection scores accumulated from all targets (zoomed-in). . . . . 73
5.5 Detection scores for each target as a function of detection boundary $\gamma_d$
(first set). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.6 Detection scores for each target as a function of detection boundary $\gamma_d$
(second set). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
5.7 Detection score differences accumulated from all targets (full domain). 77
5.8 Detection score differences accumulated from all targets (zoomed-in). 78
5.9 Detection score differences for each target as a function of detection
boundary $\gamma_d$ (first set). . . . . . . . . . . . . . . . . . . . . . . . . 79
5.10 Detection score differences for each target as a function of detection
boundary $\gamma_d$ (second set). . . . . . . . . . . . . . . . . . . . . . . . 80
5.11 Key detector efficiency. . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.12 Detection score differences accumulated from all targets for a subset of

$T_{rnd}$ (zoomed-in).

88
Chapter 1

Introduction

1.1 What is this Thesis About?

After giving a presentation on model checking in a high performance computing course, I was asked by one of my classmates—a graduate student whose research relied on computations involving Maxwell’s equations—if there was a technique that he could apply to help him find syntactically small coding mistakes—like incorrectly entered constants—in his scientific programs. The work in this thesis formulates a response to his question.

The conclusions reached in this work can be broadly summarized in two statements:

1. Provided that a tester has access to oracles that are sufficiently accurate, a small number of carefully selected tests can detect most code faults in small units of scientific code: computational scientists may be well served by modicum of testing effort focused on code.
2. Scientific software testing approaches must focus on output accuracy—not output correctness—while dealing with the fact that scientific software oracles are generally limited and imprecise: testers and testing researchers would be prudent to recognize these facts of reality.

These two thesis statements are offered in light of the fact that this work is of concern to two groups: computational scientists and software engineers.

1.2 Why is this Work Relevant?

Computational science—“the application of computing capabilities to the solution of problems in the real world”[4]—depends on knowledge, techniques, and tools that are collected from a broad set of domains that include experimental science, theoretical science, math, computer science, and electrical engineering. A growing number of professionals feel that computational scientists should add the tools and techniques of modern software engineering to this multi-disciplinary milieu, but, at this point in time, collaborations between software engineers and computational scientists are generally limited to a few specialized areas such as high performance computing.

Kelly [14] describes the separation between software engineers and scientists that has arisen since the early days of computing as a “chasm” that isolates the two groups and frustrates attempts at fruitful cooperation. When the two groups do interact, however, we find that they offer valuable, sometimes sharp, criticism of each other’s practices and principles.

For example, in the preface of [6], Bertrand Meyer—a software engineer—wrote the following:
Scientific software is a paradoxical product. The people creating it are, as one may expect, scientists.... Their titles, their credentials, just the names of their topics are enough to bedazzle the rest of us. In most cases, however, the product itself is surprisingly unscientific.

From the other side of the chasm, D. E. Stevenson—a computational science researcher—writes:

Software engineering is meant to produce software by a manufacturing paradigm, but this paradigm simply cannot deal with the scientific issues. ... I conclude that process does not develop software, people and their tools do. Second, software metrics are not meaningful when the software’s purpose is to guarantee the world’s safety in the nuclear era. Finally, the quality of simulations must be based on the quality of insights gained from the revealed science. [Emphasis is in the original.][36]

Both quotations are taken from writings that address the topic of scientific software quality, and both of them invoke science to add weight to their claims. The two quotations might even be understood to be contradictory. On one hand, by suggesting—in a textbook advocating software engineering techniques—that computational science is unscientific, Meyer seems to be implicitly claiming that computational scientists will produce software that is more scientific if they adopt techniques developed by software engineers. On the other hand, Stevenson claims that software engineering research is so far removed from science that it cannot be used to improve the quality of scientific software.

Over the course of developing this thesis, I have decided that these two comments need not be contradictory. With respect to, at least, the current state of software
testing research, both quotations may be regarded as highly perceptive. As will be discussed, the testing practices of computational scientists are very unscientific, but, as will also be discussed, many of the testing techniques that are currently offered to scientists by software engineers are unsuitable for use under the conditions in which scientific software is developed and executed.

These claims are supported by comments of practitioners who work in the criticized disciplines. In an article discussing reproducible research, a group of computational scientists echo Meyer’s sentiments:

In stark contrast to the sciences relying on deduction or empiricism, computational science is far less visibly concerned with the ubiquity of error. At conferences and in publications, it’s now completely acceptable for a researcher to simply say, “here is what I did, and here are my results.” Presenters devote almost no time to explaining why the audience should believe that they found and corrected errors in their computations. The presentation’s core isn’t about the struggle to root out error—as it would be in mature fields—but is instead a sales pitch: an enthusiastic presentation of ideas and a breezy demo of an implementation.[5]

One might expect that software engineers, who are strongly concerned with software quality, would be able to offer solutions to computational scientists who are struggling with the ubiquity of software errors. However, in the pursuit of domain independent solutions, software engineers have failed to recognize the domain specific issues that limit the applicability of these supposedly general solutions. After discussing a number of software development topics—including software testing—with 16 scientists, software engineering researchers concluded that software engineers did
not offer scientists the testing techniques that they needed:

We need research in test-case selection methods that deals realistically with the lack of oracles, computational singularities, long runtimes, large input data sets, extensive output, and software with complex domain content.[34]

The two thesis statements presented at the beginning of this chapter are each intended to address one of these criticisms. The first thesis statement is directed at scientists who often overlook the need to reproducibly test their codes, and, thereby, hurt the scientific quality and credibility of their work. The second thesis statement is directed at software engineers who, by failing to address domain specific issues, have limited the applicability of their testing solutions.

Carefully seeking to understand the offerings and deficiencies of the two groups—that is, bridging the chasm—has resulted in original research findings about the efficacy of scientific software testing as well as a novel testing technique: Mutation Sensitivity Testing (MST). The particulars of this work are focused primarily on scientific code scrutinization through testing, but, beyond concrete developments and discoveries, I believe that this thesis demonstrates the value of attentive cross-disciplinary cooperation between scientists and software engineers.

1.3 An Outline of the Work

Following this introduction:

• Chapter 2 establishes concepts and defines terms that will be used throughout the thesis. The chapter will begin by discussing errors and the model refinements
CHAPTER 1. INTRODUCTION

that introduce them. The roles of verification, validation, and code scrutinization activities will then be explored. Finally, essential code testing concepts will be identified and discussed.

- Chapter 3 presents a brief overview of literature that demonstrates the existence of scientific software code faults and the subtle but dangerous failures that they can cause. Articles that discuss and investigate current scientific software testing practices will also be examined.

- Chapter 4 will briefly describe traditional mutation testing and its shortcomings before introducing and describing the novel MST technique.

- Chapter 5 will describe MST experiments that were conducted using eight small numerical routines, four classes of mutation operators, and 1155 tests organized into three test sets. The chapter ends by presenting and discussing the results.

- Chapter 6 will summarize the thesis, discuss tenable conclusions, and suggest future studies that may be fruitful.
Chapter 2

Important Concepts

The environment in which computational scientists develop software has a number of features that must be understood by researchers who wish to help scientists improve the quality of their codes. This chapter highlights a number of these features while establishing definitions that will be used throughout this thesis. The work contained in this chapter is a synthesis of ideas and philosophies that the author encountered while working on scientific programs and while conducting research for this thesis. The concepts themselves are described in various publications, but this thesis offers a novel combination and presentation of the important considerations.

It should be noted that, in a departure from the norm, the literature review in this thesis has been postponed by one chapter (to Chapter 3). The current chapter, which contains both established and novel concepts, has been inserted in an attempt to provide context and to ensure that critical terms and concepts used in the literature review are well understood.
2.1 Accuracy and Error

Accuracy is used in this thesis to indicate the degree to which the measured or calculated value of a quantity conforms to what is considered to be its actual value.

Error—a concept related to accuracy—is defined to be a measure of the difference between a measured or calculated value of a quantity and what is considered to be its actual value. A change in error implies a proportional and opposite change in accuracy.

It is often the case that little is known about the actual value of a quantity, and, in such cases, accuracy and/or error may be estimated or bounded by studying the process and/or equipment that was used to measure or calculate the quantity.

Scientific programs analyze or simulate mathematical models of real world systems of engineering or scientific importance in order to help users better understand and predict the behaviour of these systems [35]. Insight into the behaviour of these systems is invaluable to scientists and engineers who are trying to understand and design complicated physical systems. These programs must be sufficiently accurate: if output errors are too large then scientists will not be able to develop usable insights. Note also that output inaccuracy can be dangerous: if a user does not appreciate the magnitude of the errors in a program’s outputs it may lead to poor decisions.

2.1.1 Related Concepts

It is important to understand that errors can be classified as acknowledged or unacknowledged.

Given constraints on program development and execution, acknowledged errors are either unavoidable or intentionally introduced to make a problem tractable. Sources
of acknowledged errors include simplifying assumptions, finite precision calculations, and domain discretization. In principle, the origins of acknowledged errors are fully identified [23].

Errors that result from blunders or mistakes—for example, programming mistakes, bad input data, or compiler faults—are unacknowledged errors. There are no straightforward methods for estimating, bounding, or ordering the contributions of unacknowledged errors [23].

For the sake of clarity, some other concepts that are frequently connected to accuracy should also be defined.

*Precision*, in this thesis, is used to as a measure of the exactness of a quantity. When a quantity is stored or displayed its precision is determined by the number of bits or digits that are used.

Accuracy and precision are independent measures: a number can be fully precise—that is, *exact*—but inaccurate. However, accuracy can be limited by precision. For instance, the numerical precision that is used in a calculation can affect the accuracy of the result. Acknowledged errors that are introduced when calculations are carried out using finite precision numbers are referred to as *round-off* errors.

In addition to round-off errors, *truncation errors* are a frequently encountered type of acknowledged error. These truncation errors result “from approximating continuous mathematical expressions with discrete, algebraic formulas,” [29] and do not relate to the use of finite precision numbers. Note that this usage is not fully standardized—many practitioners use “truncation error” to refer to the round-off error that is introduced when finite precision numbers are truncated rather than rounded.
2.1.2 Model Refinements Lead to Errors

Scientific code is the culmination of a series of model refinements and calculations that are broadly depicted in Figure 2.1. These refinements may involve complex transitions from one knowledge domain to another and often introduce errors. The refinements and errors shown in Figure 2.1 are described here (from top to bottom):

- Experimental science is used to take measurements of the real-world. Errors in these measurements—that is, *measurement uncertainties*—are quantified using metrological analysis.

- Theoretical science is used to formulate mathematical theories that fit and predict measurements of the real world. Errors are introduced by assumptions and approximations that are used to formulate or simplify the theories. Scientists attempt to quantify these errors by using statistical techniques to compare the predictions of a theory with measurements of the real-world.

- Numerical methods are used to formulate algorithms that can be used for the production and evaluation of predictions from theories. Truncation errors can be introduced when continuous mathematical expressions are evaluated using discrete, algebraic formulas. Truncation errors can be bounded, in principle, using numerical analysis.

- Algorithms are implemented—that is, *programmed*—in source code. Theoretically, it should be possible to implement algorithms without introducing any errors. However, implementation mistakes—referred to as *code faults*—are almost unavoidable in practice. The unacknowledged errors introduced by these code faults are studied in this thesis.
Figure 2.1: Some errors that can accumulate as a result of domain transitions.
CHAPTER 2. IMPORTANT CONCEPTS

- Compilation translates the source code into machine code that can be executed by a computer. Errors may be introduced by optimization techniques that are used to reduce the time or memory required to execute the machine code. Like truncation errors, optimization errors can be bounded, in principle, using numerical analysis.

- Machine code is executed on inputs to produce program outputs. Round-off errors may be introduced when calculations are carried out using finite precision numbers. Like truncation errors, numerical analysis can be used to bound round-off errors.

It should be noted that unacknowledged errors may be introduced in any of the model refinement steps: equipment misuse, flawed reasoning, poorly chosen algorithms, programming mistakes, faulty compilation (perhaps resulting from a mistake in the compiler’s code), and hardware faults are but a few of the possible sources of unacknowledged errors.

The error in the final output of a program is an accumulation of all errors contributed by each transition. To identify all contributions to the final error a developer or user must analyze the entire sequence of transitions.

2.2 Identifying Unacknowledged Errors

Broadly speaking, there are at least two approaches that can be used to identify unacknowledged errors and/or the faults that cause them. One approach is to carry out inspections/reviews: all artefacts that are created and/or used in the development of software can be inspected for faults. Another approach is to attempt to test/falsify
the outputs or predictions generated by artefacts for some inputs/initial conditions. In practice, these two approaches are considered complementary [8].

Scientists are familiar with review activities. Peer reviews are used extensively by scientists to assess the quality of experimental results, mathematical theories, and numerical techniques. Code inspections may, likewise, be able to help scientists improve the quality of the software that they produce; results in [16] and [9] provide some support for this hypothesis. However, the cognitive complexity inherent in most scientific codes can make it very difficult to inspect such codes thoroughly: it may be impossible to adequately comprehend the behaviour of complex numerical routines in a way that allows the inspector to discern the existence of certain types of faults. Furthermore, inspectors that are sufficiently familiar with the scientific domain, the programming language, and the numerical methods may be hard to find.

Scientists are also familiar with testing activities. Testing and falsification activities are essential to scientific research: theories are used to produce predictions that are checked against experimental results. Software testing follows a similar process: the compiled version of a code is executed on some inputs and the outputs are checked against the expected values.

In fact, one can observe that the principles of scientific falsification and software testing outlined by Popper and Myers (respectively) match up quite well—there is almost a one-to-one correspondence between principles given in [37] and [22]. In light of this observation, one might think that scientists would have naturally developed an emphasis on code testing. However, while scientists rigorously test their scientific theories, many seem to have overlooked the fact that source code should also be
rigorously tested. To understand the reason for this oversight, it is instructive to examine definitions of Verification and Validation (V&V) used by many computational scientists.

2.2.1 Verification, Validation, and Code Scrutinization

The computational science understanding of V&V is different from, but compatible with, the software engineering understanding of V&V. It should be noted that the definitions given here are not fully standardized among computational scientists, but they do currently seem to be the most popular.

Verification

Calder et al. provide the following description of verification:

Verification tests that a code or simulation accurately represents the conceptual model or intended design of the code. The process involves identification and quantification of error. Verification requires comparing the results of simulations to a correct answer of the model’s equations, which might be an analytic solution or a “highly accurate” benchmarked solution.[3]

Roache summarizes these ideas by saying that verification is about “solving the equations right.”[30]

To thoroughly verify a program, a scientist must have access to analytic or benchmark solutions for input problems that thoroughly exercise the algorithms implemented in the program. However, problem instances that have analytic or benchmark solutions may be too simple to exercise all parts of a code. For example, steady-state
problems do not, in general, exercise the full functionality of a code that solves time-
dependent problems, but one may only be able to find analytic solutions to steady-
state versions of the problems that are solved by the code. Provided that scientists
are aware of the limitations, verification tests can provide valuable information about
the suitability of the algorithms that are used by a program.

Reports suggest that verification techniques are applied inconsistently by computa-
tional scientists: there are many developers who devote only cursory amounts of
effort to verification activities. (See, for example, [23, 30, 31, 34, 33].)

Validation

Calder et al. provide the following description of validation:

Validation tests that a code or simulation meaningfully describes nature. The
process involves investigating the applicability of conceptual models.... The test compares simulation results to experimental or observa-
tional data, so validation’s scope is therefore much larger than verifica-
tion’s, requiring understanding and quantifying error or uncertainty in the
experimental results as well as in models and simulation results.[3]

Roache summarizes these ideas by saying that validation is about “solving the
right equations.”[30]

Validation tests allow scientists to quantify the accuracy of a program at a limited
number of points where experimental data is available. At any other points, the
scientists must use interpolation or (careful) extrapolation to estimate the accuracy
of the software. Complicating the situation is the fact that many scientific programs
are built to reduce the need for costly or unfeasible experiments, and, therefore,
cannot be validated for their intended usage because the relevant experimental data is unavailable. This “prediction challenge” is discussed in [28] and [23].

Scientists are familiar with the concepts and processes of validation: for hundreds of years scientists have explored methods of validating theories. Software validation is an analogous process, and, therefore, familiar to the scientists. However, not all software validations are carried out in repeatable, documented ways. This is observed by Sanders and Kelly:

Scientists use testing to show that their theory is correct, not that the software doesn’t work. ... Several [scientists] admitted that their testing practices were inconsistent, disorganized, and not repeatable. ... Still others, who admitted that their testing was unsystematic, seemed unconcerned by it, possibly because of an overwhelming focus on theory.[34]

Scrutinization

Examining these understandings of V&V, one sees that abstract theories and solution techniques, not concrete implementations, are the scientists’ primary concern. Roache’s V&V definitions from [30] are frequently quoted in computational science literature, and, in [30], Roache notes that code faults can cause significant accuracy losses. However, despite Roache’s cautions, one can find articles that have used his definitions without maintaining his concern for code correctness (for example, [30] is quoted in [3]). In 2004, Computing in Science and Engineering published a special issue on V&V: the introduction to the issue and three of the four topical articles did not mention code faults. The only article that did mention them—[31]—was written by Roache.
This thesis suggests that scientists should fill this gap in their quality assurance practices with \textit{code scrutinization} activities. That is, activities that are focused on examining code and code behaviour in order to detect faults and to determine how much realistic confidence can be placed in the software’s implementation. Some sources, such as [30] and [23], refer to these types of activities as “code verification” activities (these are distinguished from other verification activities by the prepending of “code”).

“Scrutinization” is used in this thesis in order to emphasize the fact that code and program outputs must be carefully examined in order to detect code faults. By moving away from terms like “verification,” “validation,” and “confirmation” this thesis also hopes to emphasize the fact that code scrutinization activities may be most effective when they are self-contained: that is, when they do not rely heavily on outside experimental, analytical, or benchmark data to assess code quality.

Figure 2.2 superimposes the domains of validation, verification, and scrutinization activities over Figure 2.1. This allows the reader to see the errors that can be identified and the artefacts that are (ideally) assessed by each activity. Code scrutinization activities have a more limited scope than V&V activities: one cannot use code scrutinization to check that a code implements appropriate theories or algorithms, but this limited scope tightens the focus of code scrutinization. Scrutinization activities should be more flexible because they are not limited by the available data.

Ideally, testing activities are carried out from most specific to least specific—that is, code scrutinization proceeds verification which proceeds validation. If neither algorithm verification nor code scrutinization have been carried out, a scientist finding problems in a scientific validation test would be hard pressed to decide if the source
CHAPTER 2. IMPORTANT CONCEPTS

Figure 2.2: The roles of validation, verification, and scrutinization.
of the problem is the scientific model, the mathematical model, or a code fault.

Finally, it should be noted that each of these activities (scrutinization, verification, validation) is the specialty of a different discipline: scientists are validation experts, numerical analysts are verification experts, and software engineers are code scrutinization experts. It is unlikely that computational scientists will expend much of their energy to research code scrutinization activities: “The primary goal of developing scientific software is very often to get the results of calculations from a scientific or engineering model as opposed to developing quality software for its own sake.” [33] Software engineers, as specialists in code quality, are much better suited to the task of developing scientific code scrutinization techniques provided that they recognize the unique attributes of scientific software development and execution environments.

2.3 Important Testing Concepts

The experimental work carried out in this thesis focuses on code scrutinization through testing. To facilitate the presentation of this experimental work some concepts and terms are established in this section.

2.3.1 Tests, Failures, and Faults

A code test is conducted when a tester executes a program using test inputs and then scrutinizes the outputs in an attempt to detect failures that result from code faults. Note that the occurrence of a failure does not necessarily mean that it will be detected. Also note that tests discussed in this thesis will only use legal inputs and produce legal outputs.
Illegal inputs are inputs that the developers of a program do not expect the program to evaluate. For example, a negative input would be illegal for a program that is only designed to work with positive inputs. Illegal inputs may be handled—for example, the program informs the user that the input is illegal—or unhandled.

Illegal outputs are outputs of a type that is not compatible with the expected output type. For example, if a real-valued, positive scalar is expected, then an output that is imaginary, negative, or non-scalar is an illegal output. NaN and Inf—outputs produced when a calculation involves invalid operations or an overflow—are always considered to be illegal outputs unless otherwise specified.

Tests can cause programs to fail. In the context of scientific code scrutinization tests a failure occurs when the execution of a code fault results in unacknowledged output error.

A detected failure is a failure that forces a program’s output error to exceed some detection boundary.

A silent failure is any failure that is not a detected failure.

A terminal failure is a failure that causes the executing program to produce an illegal output or to abort. Terminal failures are generally easy to detect because the outputs are unambiguously wrong, that is, they exhibit infinite error.

Code faults are mistakes in program code that can cause a program to exhibit a failure when that program is run on some legal input.

A detected fault is a code fault that has caused a detected failure. Note that a fault can be detected without being located; a detected failure implies a detected fault, but it does not necessarily provide useful information about the location and/or nature of the fault.
A *terminal fault* is a fault that can produce a terminal failure when exercised using some legal input.

### 2.3.2 Failure Detection

As detailed above, a failure can only be detected if it causes the error in a test output to exceed some detection boundary. Underlying this detection requirement is the reality that a tester must have some information about the expected output. Testing researchers refer to the source of this information as an oracle.

An *oracle* is a source—external to the program under test—that a tester can consult to determine the outputs that are expected when a test is run.

An *ideal oracle* is an oracle that provides answers with zero error for any legal input.

An *approximate oracle* is an oracle that gives an approximate value for the expected test output. Note that approximate oracles exhibit varying degrees of accuracy and precision. A domain expert who examines output from a test and declares that it “looks about right” is acting as a highly imprecise approximate oracle. In contrast, a simplified version of a complex code might be used as a highly precise approximate oracle. Note, however, that the answers provided by the simplified code might be less accurate than the expert’s opinion despite their higher precision.

*Limited oracles* are oracles that can only be used to check the outputs of some specific tests. For example, an analytic solution to a simplified problem can only be used to check that a program works accurately when the program is run using the simplified problem.

Note that an oracle can be both limited and approximate: physical experiments
are often used by scientists as limited, approximate oracles.

If (a) the software under test does not exhibit acknowledged error and (b) a
tester has access to an ideal oracle then failure detection is trivial: any outputs that
depart from the expected results are obviously incorrect. However, in some domains,
including computational science, it is often the case that (a) output correctness is
poorly defined and/or (b) available oracles are problematically imprecise and limited.
The author refers to these two problems (respectively) as the tolerance and the oracle
problems—note that these labels are novel.

The tolerance problem exists because scientific codes, by nature, exhibit acknowl-
edged errors. In general, these acknowledged errors make it impossible to determine
if an output is entirely free from unacknowledged errors for two reasons. Firstly,
in practice, it can be extremely difficult, if not impossible, to accurately bound the
acknowledged errors in an output. Secondly, even if these bounds are computed
(using fault-prone calculations) and used as tolerances, one can only guarantee the
detection of unacknowledged errors that are larger than the tolerances: smaller un-
acknowledged errors can escape detection. As a result, unacknowledged errors may
always be “hiding” in the acknowledged errors, and, therefore, one cannot speak of a
scientific program’s outputs as right or wrong, correct or incorrect.

Even if a tester has access to an ideal oracle, outputs that include acknowledged
error cannot be assessed for correctness: the tester cannot be sure that the outputs
are as accurate as they would be if no unacknowledged errors are present. The tester
can only evaluate the outputs as “good enough,” and this does not imply that the
outputs are free from unacknowledged error, only that they are free from detectable
unacknowledged error.
The oracle problem exists because it is frequently the case that developers of scientific software do not precisely know the “right” output for most, if not all, inputs—they do not have an ideal oracle. This means that scientists cannot conclusively assess their software at those points. Instead, testers must frequently rely on oracles that are limited and/or imprecise. In practice this means that scientists often rely on educated guesses, interpolations, and extrapolations that use available data and their own expertise. As a result of the uncertainty in these estimates, scientists may have to settle for a detection boundary that is much higher than is desirable.

Any techniques that are to be applied to the task of scientific code scrutinization must work despite the oracle and tolerance problems, but most testing research conducted by software engineers has been conducted under the often unstated (perhaps unrecognized) assumption that these two problems can be safely ignored. Weyuker does explicitly draw attention to the “oracle assumption” in [38], but, since the publication of that paper in 1982, the software engineering community has offered very little follow-up work. Because testing techniques developed by software engineers are often developed under oracle and correctness assumptions, the resulting techniques are of questionable value to scientists who wish to scrutinize their codes.

It is interesting to note that, because oracles and oracle interpretations can exhibit unacknowledged errors, oracle and correctness assumptions can be dangerous even in contexts where (near-)ideal oracles are the norm. With respect to the oracle assumption, Weyuker points out that when a tester misunderstands a specification it can result in “incorrect oracle output” [38]. With respect to the tolerance assumption, it should be pointed out that software testers do not always realize that acknowledged errors (such as round-off errors) are present in software outputs. For example, as
will be discussed in Section 4.2, traditional approaches to mutation testing use strict equality and this can cause problems when a program employs floating point numbers.

Other than the problematic use of strict equality in mutation testing, the considerations detailed in the previous paragraph will not be explored any further in this thesis. However, software engineers should recognize that, as a result of those considerations, some observations in this thesis may be of interest to practitioners who do not work with scientific software.
Chapter 3

Literature Review

Many scientists and software engineers do not appreciate the reality that code faults can seriously degrade the accuracy of scientific programs. Evidence—both anecdotal and experimental—of these degradations is readily available in computational science and software engineering literature. This chapter presents some signs of trouble before offering a brief survey of code fault detection techniques that are advocated and applied by some scientists and software engineers.

3.1 Signs of Trouble

Computational scientists can offer much anecdotal evidence to support the conclusion that code faults can cause scientific software errors. While working as scientific software developers, my supervisor and I both encountered accuracy degrading code faults that had previously gone unnoticed in trusted scientific codes.

Reports of scientific software code faults can also be found in the literature. In a poignant example, Dubois writes:
Small programming mistakes can ... be indistinguishable from incorrect modeling. I have spent hours in meetings looking for errors with scientists who speculated about whether some new piece of physics must be added in order to obtain a better answer. A week later, I learn that they had found a simple error [that is, code fault], and there was no scientific problem after all.[7]

The serious consequences of these faults are also illustrated anecdotally: in 2006, the discovery of a previously undetected programming mistake resulted in the retraction of five published papers and forced researchers to abandon projects that relied on the erroneous data [21].

Moving beyond anecdotes, Les Hatton and his colleagues conducted a large scale study—referred to as “the T experiments”—on the existence and impact of code faults in scientific software.

In the first part of the study, Hatton oversaw a series of static inspections of scientific codes [9]. The inspections uncovered a significant number of faults in a wide range of scientific packages. The results were not encouraging: “there are about 8 serious faults per 1,000 executable lines in C which are statically detectable in commercially released code, and about 12 per 1,000 executable lines in Fortran.”[9]

The worst offender was a nuclear engineering package that exhibited 140 weighted\textsuperscript{1} faults per 1000 lines of code: “in spite of the aspirations of its designers [this code] amounted to no more than a very expensive random number generator.”[9]

Hatton tells us that the statically inspected codes were not pre-release codes: they were subject to day-to-day use and “the developers who submitted these programs

\textsuperscript{1}Faults were weighted using an estimate of the likelihood that they would manifest themselves during execution.
believed them to be fully tested products.”[9]

In the second part of the study, Hatton teamed up with Roberts (a practicing geophysicist) to engage in a detailed comparison of nine seismic data processing packages [11]. The conclusions from this study were shocking. When testing routines that employed common algorithms—eliminating variance in abstract computational models—the nine packages only agreed to within one significant figure. Distressingly, we are told in [10] that “this [output] data is used to site oil wells and must be of at least three-significant-figure accuracy to perform this task, effectively randomizing the decision-making process.”

When standard visualizations of the outputs from any of these packages were presented, in isolation, to geoscientists they were unable to detect any significant errors, but when geoscientists were able to compare the visualizations of the outputs from the different packages they could easily spot significant differences. These sobering results were not the result of one consistently deviant package: graphs in the report demonstrate that all of the packages produce outliers.

Hatton and Roberts also conducted “feedback exercises” in which obvious discrepancies between the seismic codes were fed back to developers in an attempt to determine the reason for the discrepancies. “During the feedback exercises, suspicious departures from the average in every case were traced to a software error [that is, a code fault], and their correction led to significant reductions in disagreement.”[11]

Twelve years after Hatton published the results of these studies, he wrote a follow-up article in which he observes that, in general, computational scientists still did not appreciate the serious negative impact that code faults could have on their programs [10].
Fortunately, “code fault apathy” is not universal: explored and advocated code scrutinization techniques are discussed, to a limited extent, in computational science and software engineering literature.

3.2 Explored and Advocated Code Scrutinization Techniques

In light of the roles played by software engineers and scientists, one should not be surprised by the assertion that, in the context of computational science, software engineers advocate code scrutinization techniques while scientists apply them. Neither should one be surprised to hear that—given the disconnect between the two groups—many of the techniques advocated by the software engineers are not applied by the scientists.

3.2.1 Techniques Explored by Computational Scientists

Working against widespread code fault apathy are a small number of computational scientists who have taken their experiences and Hatton’s research to heart. In particular, some practitioners in the field of Computational Fluid Dynamics (CFD) have begun investigating code scrutinization through testing. Most promising of the resulting methods is one that is referred to as the Method of Manufactured Solutions (MMS) that is used to test Partial Differential Equation (PDE) solvers [17, 31, 30]. A very brief overview of the MMS will be given here, but, because of the conceptual complexity of the technique, this overview will not attempt to convey all details of the process: readers seeking more information should see [17].
To conduct an MMS test, a tester, first, selects a function that will be used as a solution to an input problem. The corresponding input problem is then determined by taking derivatives of the selected solution and adding necessary source terms—that is, one works backwards from the solution to the problem. The resulting input problem and its solution can then be used as a test case. When the PDE code is run on the input problem, the tester checks that the results converge to the solution at the theoretically expected rate.

Note that the MMS thus addresses the oracle and tolerance problems. By starting from the solution, the MMS is able to create complex test cases that are not restricted by the availability of analytic solutions or experimental data. By requiring that outputs exhibit theoretically determined convergence rates, the MMS is able to determine very narrow tolerances on the expected outputs.

Practitioners claim that the MMS is incredibly sensitive. Roache writes, “Upon first exposure to the MMS, most users are skeptical that it will be sensitive enough to detect all the coding errors that affect accuracy. After some experience, the complaint often changes to a charge of excessive sensitivity.”[31] Knupp and Salari tested these claims by conducting a small set of experiments in which Knupp seeded 21 faults into a code to determine if Salari would be able to detect them. All of the seeded faults that could impact the accuracy of the code were detected.

The primary weakness of the MMS is that it can only be applied to PDE solvers[31]. Admittedly, PDE solvers are present in many scientific codes, but these scientific codes also contain accessory components that may not be thoroughly exercised during a test of a PDE solver. The author has found no studies that specifically explore the impacts of possible code faults in these accessory components. A second weakness
is that thorough PDE solver testing depends on the selection of appropriate starting solutions. [17] offers guidelines on choosing solutions that will thoroughly exercise a solver code, but no controlled, large-scale studies have been conducted to demonstrate the effectiveness of these guidelines. Finally, the MMS cannot easily be applied until the PDE solver is nearly complete: the code must be able to solve the manufactured problem before a test can be carried out. Roache points out that it may take a day or two to prepare a code for testing with the MMS, and the method should be reapplied every time the code is changed [31, 30]. (Note that, because the code has already been prepared, reaplication of the method will usually take much less time than the first application.) To summarize this final weakness, one might say that the MMS lacks “agility.”

Interestingly, as Oberkampf et al. point out in [23], analogs of the MMS technique do not appear to have been discussed in software engineering literature. This may result from the fact that the MMS is only applicable to PDE solvers: perhaps the domain specificity of the method has hidden it from testing researchers?

Other than the MMS, [23] and [30] label most other code scrutinization and code fault avoidance activities—those advocated by software engineers—as “quality assurance” activities.

### 3.2.2 Techniques Advocated by Software Engineers

Software engineers have developed and documented a plethora of “traditional” code fault detection techniques. (See [2] for a comprehensive survey of many of these techniques.) Some computational scientists are aware of these techniques: for example, in [23], Oberkampf et al. refer to a number of software engineering texts on testing and
quality assurance, and, in [30], Roache discusses quality assurance activities that have been developed by software engineers. However, outside of Hatton’s T experiments (discussed in Section 3.1) and the work of Kelly et al. in [15] (discussed later in this section), the author knows of no controlled studies of these techniques that have been conducted by computational scientists.

There are very few software engineering papers that discuss the use of code scrutinization techniques in the specific context of scientific software. Two such papers—written by well known testing experts—were published in 1982. One, by Howden, provides a high-level overview of scrutinization techniques that could be used by scientists—note that Howden refers to these as “validation techniques” [12]. The other, by Weyuker, (already referenced in Section 2.3.2) highlights the fact that scientists do not usually have access to an ideal oracle, and she briefly discusses “testing without an oracle” [38]. As far as the author was able to ascertain, neither Howden nor Weyuker published any later work focused on scientific software. In 1999, Kreyman et al. released an internal report in which they discuss inspection procedures for scientific software [18]; to the author’s knowledge, this work was never published in a peer reviewed format. Finally, in a 2007 paper, Kelly et al. describe their software engineering group’s attempts to formulate a software testing strategy for a nuclear engineering research group [15].

The paper from Kelly et al. demonstrates possible benefits of collaborations between software engineers and scientists. In introduction to the paper one reads: “Scientists are the ‘caretakers’ of the models. However, a caretaker of the code is needed, and this is where the software engineer can add to the quality of the software product.” In the conclusion one finds the following:
The application of the [testing] strategy [discussed in the paper] showed how well know-how from the two disciplines, that of the software engineer and that of the professional end-user developer [that is, the scientist] could be meshed to achieve a better software product. ... The help provided by the software engineer proved to be far more extensive, systematic, and fundamental than expected. The software engineer however could not do this alone. The close cooperation of the two disciplines was key.

The success of this collaboration between scientists and software engineers suggests that modified versions of traditional code scrutinization techniques can be successfully applied to scientific software. Coincident to this observation we find the following in [33]:

Effective and efficient testing methods and techniques specifically developed for scientists have not been put into the scientists’ hands. The interlocking risks influencing the testing of scientific software means that testing strategies cannot be directly imported from other domains. The building blocks of effective testing strategies likely exist, but how to put them together in a way that meets the goals of scientists poses a unique challenge. There is some important and critical work that could be done here.

If scientists are to benefit from the expertise of software engineers, software engineers must do more than advocate code fault detection techniques: they must adapt them to, and assess them in, the environment of computational science.
Chapter 4

Code Mutation

After acknowledging oracle and correctness assumptions, testing researchers can begin to develop methods that assess the viability of various test techniques in the context of scientific software development. The author’s development of one such method—Mutation Sensitivity Testing (MST)—is described in this chapter.

4.1 Traditional Mutation Testing

A technique known as mutation testing has been used by software engineers for over 30 years and has many desirable features (see, for example, [27]):

- It can be used to evaluate sets of tests in the absence of an oracle.
- It is a well established and studied process.
- It is a technique that is largely automated.
- Its results are repeatable and it leaves a paper trail—that is, it supports reproducibility.
To discuss mutation testing it is necessary to establish some terminology:

- **Mutation**: a syntactic change to a program statement.

- **Mutation Operator** ($\phi$): a rule that is applied to program code to generate mutations. A set of mutation operators is denoted by $\Phi$.

- **Mutation Target** ($P_0$): a program that is to be mutated.

- **Mutant** ($P_m$): a program identical to some $P_0$ except that one of its statements contains a mutation. A set of mutants generated by applying $\Phi$ to $P_0$ is denoted by $\Phi(P_0) = P_M$.

- **Test** ($t$): inputs for $P_0$; $P(t)$ indicates the set of outputs produced by $P$ on $t$. A set of tests is denoted by $T$.

- **Viable Mutant**: if there exists some legal $t$ such that $P_m$ does not terminally fail when it is run on $t$, then $P_m$ is said to be viable.

- **Equivalent Mutant**: if $P_m(t) = P_0(t)$ for all $t$ then $P_m$ is said to be equivalent to $P_0$.

For later use in this thesis, another type of mutant will be also defined: *practically equivalent mutants* are mutants that are not necessarily equivalent, but which may be considered equivalent for practical purposes. A mutant $P_m$ of $P_0$ must meet at least one of the three following conditions to be classified as practically equivalent: (1) it is equivalent, (2) it is equivalent provided that it is not run on inputs that would be illegal for $P_0$, or (3) given the application domain in which the program will be used, $P_m$ contains a mutation that results in a different, but still valid, implementation of $P_0$. For example, a mutation of the `nwsqrt` function (see Section 5.2) changes
\[ \text{abs}(y^2 - x) > \text{tol} \]

to

\[ \text{abs}(y^2 - x) \geq \text{tol} \]

Both versions of this statement can be used in acceptable implementations of the Newton method algorithm, and therefore, this mutation can be considered to be practically equivalent. Note that this third condition is subject to the judgement of a domain expert.

Mutation operators are usually conceptually simple rules that modify code in straightforward ways. Common mutations found in mutants include deleted statements, negated conditionals, and replaced operators (e.g., an addition operator may be replaced with a subtraction, multiplication, or division operator).

It is important to understand that a mutation test does not, directly, reveal the presence of any code faults. Strictly speaking, mutation testing does not test code, it tests tests. Mutation testing reveals the “holes” in a test set. By choosing new tests that plug these holes, the tester reduces the number of mutants, and hence faults, that the test set cannot detect.

A test set \( T \) for \( P_0 \) can be improved using mutation testing as follows:

1. \( P_M \) is generated from \( P_0 \) by applying some \( \Phi \).

2. \( P_0 \) and every \( P_m \) in \( P_M \) are run using \( T \). If \( P_m(t) = P_0(t) \) for all \( t \) in \( T \) then \( P_m \) is a survivor; otherwise, \( P_m \) is said to have been killed.

3. Surviving mutants are examined in an attempt to determine if any of them are equivalent mutants; equivalent mutants can be ignored because, from the perspective of the user, they are identical to the original program.
4. If there are any non-equivalent (or potentially non-equivalent) survivors then $T$
   is augmented in an attempt to kill these survivors.

5. Steps 2 to 4 may be repeated using augmented versions of $T$ until the tester is
   satisfied with the results.

Note that during the second step of this mutation process, the mutation target is
used as a pseudo-oracle. In effect, a mutation test determines if a test set is able to
differentiate some “incorrect” versions of $P_0$—that is, some non-equivalent mutants
of $P_0$—from $P_0$. Mutation testing, therefore, only circumvents the oracle problem
with respect to test set evaluation. To use a mutation evaluated test set to assess a
program, a tester must still have access to an oracle for that program.

To apply mutation testing to the problem of “real world” code testing it is neces-
sary to introduce another concept: a set of mutants $P_M$ can be said to be representa-
tive of a set of faults $F$ if tests that kill all the mutants in $P_M$ also detect all the
faults in $F$. A set of mutation operators is said to be representative if it generates
mutants that are representative of real world faults.

Experimental work has shown that, in practice, a test set that kills most mutants
generated by a representative set of mutation operators can kill most code faults that
were found in real world codes [1, 25, 24].
4.2 The Deficiencies of Traditional Mutation Testing

Unfortunately, these findings cannot be applied to scientific codes because mutation researchers have held the correctness and oracle assumptions. To the author’s knowledge, all mutation testing research conducted thus far applies the rule that a mutant is killed when it produces outputs that are strictly not equal to the outputs of the target program. The following example illustrates why the use of this rule makes mutation testing unsuitable for evaluating tests of codes that employ floating-point numbers.

Consider a line of MATLAB code:

\[ y = 1 - \sin(x \times 2\pi); \]

This line can be mutated to produce

\[ y = 1 - \sin(x \times 1\pi); \]

Using \( x = 1 \) to test these two statements (on the author’s computer) yields the following results:

- Original Code: \( y \approx 1 + 2.2204 \times 10^{-16} \)
- Mutant: \( y \approx 1 - 1.1102 \times 10^{-16} \)

Note that neither of the answers are strictly correct: exact calculations, in both cases, would produce the correct result of \( y = 1 \). Also note that the mutant produces a more accurate result for this input, but, because the answers are not equal, the mutant is killed by this test.
There are two reasons why this kill result should be considered dangerous. Firstly, the error exhibited by the mutant could be made much more apparent by using a different test. For instance, using $x = 0.7$ produces an absolute difference between the two results that is over $10^{15}$ times larger. The assumption that it is “good enough” to make a mutant exhibit small errors may result in test sets that do not strongly reveal faults. Secondly, when using exact arithmetic, the two sample statements are equivalent for the $x = 1$ test: this means that the kill result is entirely due to round-off errors. An alternate test could force the non-equivalence of these two results to be properly demonstrated. When a small perturbation in round-off error is enough to kill a mutant it makes it easy to kill mutations in numeric code, and that, in turn, may lead to false confidence in the adequacy of a test set.

In effect, operator representativeness and mutation killing is complicated by the fact that strict equality is unsuitable. By using tolerances instead of strict equality one can attempt to salvage the situation, but this shifts the goal posts and forces one to question the applicability and even validity of past mutation research.

### 4.3 Mutation Sensitivity Testing (MST)

To address these problems, the work in this thesis has used a modified mutation process that avoids strict correctness testing. Instead of checking that $P_m(t) = P_0(t)$, this novel Mutation Sensitivity Testing (MST) process measures mutation sensitivity: it examines the output changes that result from the introduction of mutations. In this work, this is done by generating mutations for eight functions and then examining the relative change—that is, the relative error—in the outputs.

Sensitivity results can be used for a number of purposes. For example, one might
use sensitivity scores to quantify the strength of a test set or testing technique with respect to specific mutations, to explore errors introduced by certain classes of mutations, or to help a tester choose appropriate detection boundaries.

MST is virtually untried as a test and fault analysis tool: this thesis hopes to lay the groundwork for more comprehensive assessments of MST. To that end, it will be helpful to establish some additional concepts and definitions.

### 4.3.1 MST Definitions

- **Relative Error** ($\gamma$): the relative error for a given mutant $P_m$ and test $t$ is
  \[
  \gamma(P_m, t) = \frac{|P_m(t) - P_0(t)|}{|P_0(t)|}
  \]
  where $P_0$ is the function that was mutated to produce $P_m$. Note that relative error is ill-defined if $P_0(t) = 0$. In this work: if $P_m$ fails terminally then the relative error is defined to be infinite; if a relative error calculation overflows for a legal value of $P_m(t)$ then the error is defined to be the largest, non-infinite double precision number (that is, approximately $1.7977 \times 10^{308}$). (These rules allow a tester to distinguish very severe non-terminal failures from terminal failures.)

- If $P_0$ and $P_m$ output $n$ numerical values then the $L^\infty$-norm can be used, that is,
  \[
  \gamma(P_m, t) = \max_{1 \leq i \leq n} \left\{ \frac{|P_m^{(i)}(t) - P_0^{(i)}(t)|}{|P_0^{(i)}(t)|} \right\}
  \]
  where the bracketed superscripts indicate the $i$th output from $P_m$ and $P_0$.

- **Maximum Exhibited Error** ($\Gamma$): the maximum exhibited error $\Gamma$ for a given $P_m$ and set of tests $T$ is
  \[
  \Gamma(P_m, T) = \max_{t \in T} \{ \gamma(P_m, t) \} 
  \]
In other words, $\Gamma$ indicates the largest relative error exhibited by a mutant that is tested using a set of tests.

- **Revealed Mutant**: a mutant $P_m$ is revealed by a test set $T$ if $\Gamma(P_m, T) \neq 0$.

- **Strongest Revealers**: given $T$, the strongest revealers for a given set of mutants $P_M$ are
  \[ \{ t \in T | \gamma(P_m, t) = \Gamma(P_m, T) \text{ for some } P_m \in P_M \} \]
  In other words, the strongest revealers are those tests that cause some mutant to produce its maximum exhibited error.

- **Maximally Revealed Mutants**: given $P_M$, the mutants that are maximally revealed by $t$, $t \in T$ are
  \[ \{ P_m \in P_M | \gamma(P_m, t) = \Gamma(P_m, T) \} \]
  In other words, the mutants that are maximally revealed by a test are those mutants for which the test is a strongest revealer.

- **Minimal Set of Equivalent Tests**: given $T$ and $P_M$, the minimal set of equivalent tests $T_Q$ is a minimal subset of $T$ such that
  \[ \forall P_m \in P_M, \quad \Gamma(P_m, T) = \Gamma(P_m, T_Q) \]
  In other words, a minimal set of equivalent tests is a minimal subset of $T$ that generates the same $\Gamma$ scores as $T$.

- **Detected Mutants**: given $\gamma_d$, $P_M$, and $T$, the mutants detected by $T$ are given by
  \[ D(\gamma_d, P_M, T) = \{ P_m \in P_M | \Gamma(P_m, T) \geq \gamma_d \} \]
In other words, detected mutants are those mutants that exhibit error larger than or equal to $\gamma_d$ when tested using $T$; they are the MST analog of killed mutants.

- **Detection Score**: the detection score of $T$ is the fraction of mutants detected by $T$ for a given $P_M$ and $\gamma_d$, that is,

$$\frac{|\mathcal{D}(\gamma_d, P_M, T)|}{|P_M|}$$

In this thesis, the detection score is calculated using only mutants that are viable and not practically equivalent. Detection scores are usually presented using percentages.

- **Test Set Efficiency**: given $\gamma_d$ and $P_M$, the efficiency of a test set $T$ is given by

$$\frac{|\mathcal{D}(\gamma_d, P_M, T)|}{|T|}$$

In other words, the efficiency of a test set is the number of mutants that are detected per test.

- **Key Detectors**: given $\gamma_d$, $T$, and $P_M$, the set of key detectors $T_K$ is a minimal subset of $T$ such that

$$\mathcal{D}(\gamma_d, P_M, T) = \mathcal{D}(\gamma_d, P_M, T_K)$$

In other words, the set of key detectors of $T$ is the minimal set of tests from $T$ that can detect all the mutants that are detected by $T$.

- **Tolerable Mutant**: a mutant $P_m$ is tolerable with respect to $\gamma_{MAX}$ if

$$\forall t, \gamma(P_m, t) \leq \gamma_{MAX}$$
where $t$ is any legal input for the program $P_0$ that was mutated to produce $P_m$. Tolerable mutations are the MST analogue of equivalent mutants. A mutant that cannot produce a relative error in excess of $\gamma_{MAX}$ is of little to no concern in a context where $\gamma_{MAX}$ is a negligible error.

**Some Notes on Detection Boundaries**

In this work, the detection boundary $\gamma_d$ is left as a flexible analysis parameter, but in practice, it may be necessary to specify a fixed $\gamma_d$ for every test that is applied to a program. That is, to judge a test, a tester must be decide if the error in the output is acceptable—some detection boundary is explicitly or implicitly used to make this judgement. Ideally, estimates of oracle accuracy and unacknowledged error would be used to carefully determine explicit detection boundaries, but, in actual practice, detection boundaries are often implicitly and unreproducibly applied when an expert judges that the output “looks about right.” The principles underlying MST emphasize the importance of carefully choosing and documenting the failure detection criteria that are used when programs are tested.
Chapter 5

MST Experiments

The experiments presented in this thesis were conducted to show how one might go about using MST—in this case, sets of tests are applied to mutations of different targets and the errors in the results are analyzed to assess the efficacy of the different test sets.

The MST process described in this thesis can be seen as a function (i.e., a “black box”) that takes in a set of mutation operators, a mutation target, and a set of tests—possibly selected using a specified testing technique—and outputs aggregate information about the accumulated results. The experiments described in this chapter are conducted by holding two of the MST inputs constant while varying the remaining one.

It should, however, be noted that test set selection depends on properties of the mutation target, and, therefore, a concrete set of tests cannot be held fixed while the mutation target is varied. In order to overcome this limitation, the test input is abstractly viewed as a testing technique rather than a fixed set of tests.

The first section of this chapter will describe the mutation operators and then
briefly survey the software (MATmute) that was developed to apply them. The second section will describe the mutation targets that were used. The third section will describe the test sets that were applied after describing the techniques that were used to select them. The fourth section will describe the analysis of the error results and present some pertinent graphs. Finally, the fifth section will discuss the results.

5.1 The Mutation Operators and MATmute

For these experiments, 1492 mutants were produced; this section discusses the mutation operators and software used to produce and execute these mutants.

5.1.1 The Mutation Operators

The following set of mutation operators—adapted from [1]—was used for all the tests:

- **Statement Deletion**: deletes a statement—results in one mutant per statement.
- **Conditional Negation**: logically negates the conditional part of an if or while statement—results in one mutant per conditional statement.
- **Constant Replacement**: replaces a hard-coded constant $C$ with $0$, $-C$, $C - 1$, $C + 1$, $0.9C$, and $1.1C$—results in six mutants per hard-coded constant unless $C \in \{-1, 0, 1\}$ in which case only non-duplicate mutants are created.
- **Operator Replacement**: replaces an arithmetic ($+$, $-$, $*$, $/$, $\backslash$, $\wedge$), relational ($<$, $<=$, $>$, $>=$, $==$), or logical ($\&\&$ and $||$) operator with another operator of the same class—results in five mutants per relational operator, five mutants per arithmetic operator (except for $+$ and $-$ which result in nine mutants because
they must be replaced by both matrix and element-by-element operators), and one mutant per logical operator.

In [1], Andrews et al. statistically compare mutant behaviour with actual fault behaviour to demonstrate that these mutation operators are representative of faults that are found in real world software. Unfortunately, because mutant detection/killing criteria have changed, operator representativeness established using traditional mutation testing does not hold, in general, when the same operators are used for MST or for scientific software testing.

Defining, much less exploring, representativeness in the context of MST appears to be a very difficult problem, and, for that reason, no attempts were made to show that the mutation operators selected in this work are representative of real-world faults. Research that explores the properties of these and other mutation operators in the context of MST may be valuable.

Having established this caution, there is no avoiding the fact that mutation operators are essential to the production of mutations: a traditionally representative set of mutation operators was chosen as an intuitively reasonable starting point. Also note that, while it is almost certain that the selected mutation operators are not fully representative of real world faults, it is certain that they are representative of, at least, those real-world code faults that are identical to the mutations that are introduced.

5.1.2 MATmute

All mutants were created and executed using the MATmute software package developed by the author.\textsuperscript{1} The software uses a collection of Python modules and MATLAB

\textsuperscript{1}The current version of MATmute—including the source code—is available at matmute.sourceforge.net.
functions: Python is used to generate the mutants while MATLAB is used to execute them and analyze the results. Complete code listings for the software can be found in Appendix A.

The Python Modules

MATmute generates mutants—that is, it applies mutation operators—using a 738 line (excluding comments) Python program that is divided into 12 modules.

The mutant production process broadly follows the following outline (in this outline, a monospaced font is used to indicate a Python module):

1. `matmute` asks `mutatee` to read and store the statements from the target code; `mutatee` stores the results as *instrumented code* (see below).

2. `matmute` creates an `operators` object that is set to use the mutation operators that were selected in the input.

3. `matmute` tells `mutants` to use the `operators` object to get and store mutations of the target statements.

4. `mutants` uses `mutator` to ask `operators` for the mutations.

5. `operators` produces mutations as configured in `ops_config` using the selected operators.

6. `matmute` asks `mutants` to generate the mutant files and save them to a `mutes` directory; `mutants` also stores a copy of the instrumented code (see below) in this directory.
When possible, a language independent implementation of this mutation process was attempted in the hopes that it would lead to a well organized and adaptable code. However, some parts of the code—particularly those that involve code parsing and code generation—had to be built in a language specific way.

A MATLAB grammar or stand-alone parser could not be obtained so all code parsing was done using string methods and regular expressions. Pattern matching, find-and-replace string operations were sufficient to generate all the mutations that were examined in this thesis, but the use of more advanced parsing techniques would allow testers to generate more complex and/or context aware mutations. Improved contextual awareness could be used to reduce the number of non-viable mutants and/or equivalent mutants that are produced by MATmute.

**Instrumented Code**

The *instrumented code* is a version of the target code that has been instrumented as follows: (1) comments have been removed (except for MUTE comments that can be used to turn mutation on or off), (2) statements have been split and/or consolidated so that there is exactly one statement per line, (3) loop monitoring code has been inserted (if applicable—see below), and (4) MUTE comments have been inserted to prevent function declarations from being mutated. For any inputs, the outputs produced by the instrumented function should be identical to the outputs produced by the target function. MATmute checks that this is true for every test that is executed.
Loop Monitoring

Mutations can introduce faults that lead to infinite loops. Obviously, these infinite loops cannot go unchecked or it would be impossible, in general, to run tests on a full set of mutants. MATmute addresses the occurrence of infinite loops by instrumenting all loops in a target code and its mutants with loop monitoring code. MATmute records, for every test, the number of times that the loops in the instrumented version of the code are executed. These values are then incremented by 1 (to ensure that they are non-zero) and multiplied by some value that is larger than 1—for this thesis a multiplier of 100 was used; the results are then stored in a global array (\texttt{tick\_limit}) that can be accessed while the mutants are executing.

At the start of every loop iteration, the loop monitoring code of an executing mutant increments the loop count and then checks to make sure that the count is less than or equal to \texttt{tick\_limit}. If the count is greater than \texttt{tick\_limit} then the mutant terminates itself and MATmute records a relative error of infinity for that test-mutant pair. This approach could also terminate non-infinite loops, but only after the number of loop iterations has exceeded \texttt{tick\_limit}. If we assume that a user would notice faults that degrade performance by a factor of, approximately, \texttt{tick\_limit} (100 in this work) then this premature termination is acceptable.

The MATLAB Functions

The Python portion of MATmute can be run outside the MATLAB environment, but the mutants that are produced must be executed in MATLAB. MATLAB, an environment designed for numerical analysis, is also well suited to the task of analyzing the mutants’ outputs, and therefore, MATmute executes and analyzes generated
mutants using a collection of MATLAB functions written by the author.

The following MATLAB functions are most important to this research:

- **get_error**: given two inputs, this function uses the first input as a reference value to calculate the relative and absolute errors in the second input.

- **getapproxsetcover** and **getminsetcover**: given a universe (that is, a set of integers) to be covered and a set of subsets of the universe that can cover the universe, these functions output, respectively, an approximate-minimal or minimal list of the subsets needed to cover the universe. These two functions are used by other functions as detailed below.

- **getkeykillers** and **getkeytests**: given a matrix of relative error values, these functions output, respectively, the key detectors or the minimal set of equivalent tests.

- **matmute**: given a target function, a set of tests, and some other optional input parameters, this function coordinates the generation and execution of mutants. It outputs the relative error for each mutant-test pair, the relative time taken by each mutant-test pair, the output from each mutant-test pair, and the original function’s outputs for each test.

From the above list, only **get_error** and **matmute** are required to produce MST results, the other functions, however, are used to carry out the analysis that is presented in this thesis.

The following dependencies exist among the functions listed above:

- **matmute** uses **get_error** to compute the relative error for each test-mutant pair.
• *getkeytests* transforms the problem of finding equivalent tests into a problem of finding the key detectors and then uses *getkeykillers* to find a solution.

• *getkeykillers* uses *getminsetcover* to determine the minimal set of tests that detect (that is, cover) all detected mutants.

• *getminsetcover* uses *getapproxsetcover* to reduce the time to solution by using the size of the approximate solution as a bound on the size of the optimal solution.

Note that the sensitivity measure that is used to assess the mutant’s outputs can be changed by replacing *get_error* with a function that computes a different measure. Only relative error results calculated by *get_error* were used in this work.

**MATmute Code Faults**

The author does not believe or expect that MATmute is free from code faults. However, specific design philosophies and development practices were employed in an attempt to mitigate code fault related risks.

It should be noted that errors in the outputs of the Python modules—that is, syntactic errors in the mutants—would impact the reproducibility of these results, but such errors would not invalidate the results. That is, the generated mutants would be different than those that were expected, but, because representativeness is not required, these mutants would still be sufficient for the exploratory research that is discussed in this thesis. Having said that, a number of mutants were manually inspected during the course of the thesis research, and they were found to match expectations.
Significant errors in the outputs of the MATLAB functions, on the other hand, would impact the validity of the results discussed in this thesis. The oracle problem complicates the testing of these functions: it is difficult to predict how a large set of mutants will behave when conducting MST tests. Test results had to be inspected for consistency or compared with results from an alternate implementation of the function under test. Any suspicious outputs from the MATLAB functions led to careful scrutinization. For example, an MST test of one target function resulted in no mutants that were non-terminally revealed—this suspicious behaviour was traced to a fault in the way that get.error handled some cell arrays.

Code readability and simplicity were considered to be of paramount importance during the development of MATmute. When forced to choose between execution speed and conceptual complexity, the author generally preferred the implementation that was, in his opinion, more conceptually simple. In one case, however, the author could not follow this general rule: the getminsetcover function exhibits conceptual complexity that had to be added in order to achieve a reasonable execution time.

Python code was built in a modularized way to avoid intricate webs of control flow. Objects oriented methods were employed when convenient, but a procedural paradigm was generally employed to match the procedural nature of mutant generation and analysis processes.

Small tests were frequently employed during development. An attempt was made to achieve a statement coverage level of, at least, 90%, but the quality of these tests was difficult to assess. The Python code employed a number of regular expressions that were difficult to exercise, and the numerical nature of the MATLAB functions made it difficult to assess test quality for the reasons discussed in this thesis.
A "do-it-twice" approach (see [32]) was used in the construction of many of the modules and functions. Test results from the second-time implementations were checked against the prototype implementations to ensure consistency—a number of minor and serious faults were identified this way.

Static code analyzers (Pylint [19] and M-Lint [20]) were run on the codes and critical warnings were addressed. Kelly and the author then printed out and inspected the codes. These static inspections detected some minor faults.

Assertions were included in the getkeytests and getkeykillers functions to check that the test sets returned by these functions were, in fact, able to (respectively) maximally reveal or detect all mutants that were maximally revealed or detected. As a result of these assertions, the author identified some faults in the getminsetcover function.

When feasible, final analysis steps were conducted in redundant ways in attempt to verify the analysis results. For example, detection scores at various $\gamma_d$ values were calculated both by using a simple (one line) detection counting routine and by using the MATLAB histc and tabulate functions.

The author feels that these philosophies and practices helped him significantly improve the quality of his code. MATmute is not flawless, but it believed that the results in this thesis can be trusted.

5.2 The Mutation Targets

Eight mutation targets and their mutations were studied:

- **binSearch** (15 statements): conducts a binary search for a numerical value in a vector.
• **gaussQuad** (18 statements): uses Gaussian quadrature to integrate a function.

• **GEPiv** (24 statements): uses Gaussian elimination with partial pivoting to solve a system of equations.

• **nwtsqrt** (6 statements): uses Newton’s method to find the square root of a number.

• **odeRK4** (15 statements): uses the fourth-order Runge-Kutta method to solve a single, first-order ordinary differential equation (ODE).

• **powerit** (9 statements): uses the power method to find the largest eigenvalue of a matrix.

• **simpson** (6 statements): uses Simpson’s rule to integrate a function.

• **sphereFnet** (15 statements): computes the forces on a sphere falling through a fluid.

Full code listings can be found in Appendix B. Informal input/output specifications for the functions are provided in Section 5.3.2.

These eight MATLAB functions were taken from an introductory scientific computing course. They are smaller than most scientific codes from the real-world. However, they are all, with the exception of **sphereFnet**, meant to be used as components or units in larger scientific codes, and they all contain code statements of the type that are essential to numerical analysis.
5.2.1 Modifications for MST

The functions were modified slightly (by hand) from the original versions to make them more amenable to MST:

- All printing statements were removed to keep the standard output clear of unnecessary text and to improve performance. This modification should not affect values returned by any of the functions.

- Some statements that dealt with illegal inputs were removed to eliminate the large number of practically equivalent mutants that result from the mutation of these types of statements. Given that our research focuses on legal inputs, this change does not detract from our results.

- The `gaussQuad` function was modified so that node weights were always computed. The original version used a hard-coded look-up table to determine node weights when 8 or less nodes were used, and, because mutations of this look-up table can be seen as data mutations rather than code mutations, it was decided that this look-up table code would be removed from consideration. In future studies it may be instructive to examine how these data mutations affect program behaviour.

- The tolerance inputs for `GEPiv` and `nwtsqrt` were replaced with hard-coded tolerances. The `GEPiv` tolerance is used to prematurely terminate execution when a near-zero pivot value (that is, a pivot that is less than the tolerance) is encountered. The `nwtsqrt` tolerance is used to determine when the solution has been found to a sufficient precision. Because these tolerances act as hard tolerances on accuracy, it was decided that they should be kept constant during
MST in order to ensure fair comparisons of error results from different tests. Hard-coding the tolerances ensures that they are not accidentally modified and exposes them to controlled mutations.

- The odeRK4 outputs were changed so that the vector of independent variable sampling points was not returned by the function. This vector of sampling points is generated at the beginning of the function—it is only vulnerable to two mutations—and—because this vector is used in a number of calculations—errors in this vector will manifest themselves in the solution vector. Therefore, the vector of sampling points was removed from the output to simplify the output.

- Iteration seed and shift parameter inputs for the powerit function were replaced with hard-coded constants. Changing these parameters can have an impact on the convergence rate of the function and this can complicate error trends that are seen in the outputs. Therefore, these inputs were changed to hard-coded constants to simplify the MST analysis.

It should be noted that MST could be applied without making these changes, but these changes reduce the number of equivalent mutants produced while simplifying the selection of tests and analysis of the results.

### 5.3 The Tests

1155 tests were selected in total: 105 Popperian tests and 1050 pseudo-random tests. This section describes the testing philosophies that underlie these classes of tests before describing, in detail, the test selection procedures for each mutation target.
5.3.1 The Testing Philosophies/Techniques

Popperian Testing

Based on the personal experiences of Kelly and the author, the author pioneered a testing approach that is loosely based on techniques such as boundary value testing and equivalence class partitioning (see, for example, [13])—the author refers to this technique as *Popperian testing*.²

Popperian tests should be selected to encourage novel computations that push the boundaries of a program in attempt to falsify its code. These tests are not meant to prove partial code correctness—that is a task for formal mathematics—nor are they meant to push the boundaries of the science—that is task for validation activities. Instead, these tests are designed to expose faults in a code analogously to the way that some experiments are designed to expose flaws in a theory.

The Popperian approach to testing should not be seen as a formalized testing technique. It is often difficult to clearly define the domain or boundaries of scientific programs, and, therefore, testers must rely heavily on their intuition and domain expertise when choosing Popperian tests. Again, this is analogous to scientific falsification: in most cases, scientists do not have formal guidelines that instruct them how to choose falsification experiments. Instead, scientists must rely on their expertise and intuition to guide them in the production and testing of novel predictions that are likely to falsify a theory.

Note that Popperian tests avoid the use of illegal inputs: it has been found that these *boundary exceeding* inputs are not helpful when conducting sensitivity tests.

²Karl Popper was a philosopher of science who used falsifiability as his criterion for demarcating science from non-science [37]. Popperian testing has been named in recognition of the influence that Popper’s work has had on the author’s approach to scientific software testing.
because they usually lead to nonsense outputs (e.g., NaN or Inf) and often do not exercise very much of a code. That is not to say that such test inputs are not of value in other contexts: they are needed in traditional boundary value tests and may be very important to ensure that software misuse is handled safely.

To illustrate how the Popperian approach works, consider a program that, given a function \( f(x) \) and two double precision values \( a \) and \( b \), uses Simpson’s rule to numerically approximate
\[
\int_a^b f(x)
\]
The traditional boundary value testing approach requires that the testers determine bounds for each input, but appropriate bounds for \( f(x) \) are not obvious. The problem is further complicated by the fact that the boundaries of \( a \) and \( b \) are determined by the \( f(x) \) that is used. When confronted with this problem, the author recalled that Simpson’s rule uses second order interpolants and should, therefore, exhibit different behaviour when moving from a cubic polynomial to quartic polynomial. Therefore, it was decided that the function would be tested using \( f(x) = x^3 \) and \( f(x) = x^4 \). A and \( b \) where then selected so that the outputs would be both small and large and so that the computations would involve both negative and positive numbers. In short, the author drew on expertise and intuition to select appropriate falsification tests.

Because Popperian test selection relies on intuition, the test selection approach used for these experiments may be difficult to reproduce. However, since this preliminary work is focused primarily on MST and not on the development of testing techniques, it was felt that this lack of reproducibility was acceptable. If further work is to be done in this area it may be important to better define the Popperian approach.
Pseudo-Random Testing

For comparison with Popperian testing, tests were selected using a pseudo-random method. Numerical inputs were selected by using the MATLAB `rand` function to select random values from defined intervals of reasonable values—note that the Popperian test inputs were taken from the same intervals. For non-numerical inputs—e.g., the function to be integrated by the `simpson` routine—the Popperian test inputs were used.

5.3.2 Test Details

Three variables were used to represent the three sets of tests used in this research:

- $T_{Pop}$: the 105 Popperian tests.
- $T_{rnd}$: the 1050 pseudo-random tests.
- $T_{cmb}$: the set of 1155 tests that is formed by combining the Popperian test set with the pseudo-random test set.

For the sake of brevity, when discussing a specific mutation target, the above symbols may be used, but it should be noted that, in that context, the symbol is only meant to indicate the tests from the indicated set that are designed for the target function that is being discussed. For example, if discussing the testing of `binSearch`, the symbol $T_{Pop}$ indicates the Popperian tests that were selected for `binSearch`.

The input and output specifications for each mutation target are described below. Noteworthy input, output, and behavioural constraints are also given—if these constraints are violated the inputs and/or outputs are considered to be illegal. Finally, the tests selected for each target are described.
When inputs are described as small, nominal, and large, it has a technical meaning: the nominal values for each input were used for all tests except that one test was run for each input where that input was set to the small value and one test was run for each input where that input was set to the large value. For example, if a function takes two inputs and, in the test descriptions, both inputs are described as being set to 0 (small), 1 (nominal), and 2 (large) then five tests would be created: \( t_1 = (1, 1), t_2 = (0, 1), t_3 = (2, 1), t_4 = (1, 0), t_5 = (1, 2) \). Tests inputs that are not described using small, nominal, and large values do not follow this pattern.

All tests used in this thesis research were selected so that \( P_0(t) \neq 0 \) in order to avoid complications due to the ill-defined nature of relative error (\( \gamma \)) for reference values of 0.

All tests were selected before examining the mutant outputs with the exception of two feedback tests that were added after it was found that five unrevealed mutants (four from \texttt{GEPiv} and one from \texttt{sphereFnet}) could be revealed by the addition of two tests. These feedback tests are explicitly noted in the test set descriptions.

Note that \( \epsilon \) is used to represent machine epsilon for double precision numbers (approximately \( 2.2204 \times 10^{-16} \)).

\textbf{binSearch}

\textbf{Inputs}: (1) a vector \( X \) of monotonic reals and (2) the real value \( r \) that is to be located.

\textbf{Output}: the index \( i \) in the vector such that \( X(i) \leq r \leq X(i + 1) \).

\textbf{Constraints}: \( \min(X) \leq r \leq \max(X) \).

\textbf{\( T_{Pop} \) (11 tests)}: Three \( X \) vectors were created by randomly selecting 23 values
from \([-1, 1], [-\epsilon, \epsilon], \) and \([-10^{10}, 10^{10}]\) (one range was used for each vector). The ends of the ranges were then appended as the start and end points of the vectors. The \(r\) values were then selected so that each vector was searched for the end points and 0. The two final tests were created where the length of \(X\) was 1. For the first of these two final tests, both inputs were 1; for the second, both were 10^{10}.

**\(T_{\text{rnd}}\) (110 tests):** For these tests, the length of each \(X\) vector was randomly selected from integers in \([1, 25]\). The elements filling each \(X\) were randomly selected from \([-10^{10}, 10^{10}]\) and sorted in ascending order. Each \(r\) value was randomly selected from \([\min\{X\}, \max\{X\}]\).

**\texttt{gaussQuad}**

**Inputs:** (1) the function to be integrated, (2) the upper bound of integration, (3) the lower bound of integration, (4) the number of panels used for integration, and (5) the number of nodes to use in each panel.

**Output:** approximate value of the integral.

**Constraints:** the function must be defined at any point between and including the integration bounds, the integration bounds must be real, and the number of panels and nodes must be positive integers. Output must be scalar and real.

**\(T_{\text{Pop}}\) (18 tests):** two nodes per panel were used for most tests—the usage of two nodes means that these tests will exhibit no truncation error if a polynomial of order 3 or less is integrated. Therefore, in order to produce outputs both with and without truncation errors, a cubic function and a quartic function were integrated. Integrations limits were all selected from \([-10, 10]\). Near-symmetric and non-symmetric integration limits as well as limits that included roots were selected in order to ensure
that the calculations would deal with negative, positive, and near-zero values. The number of panels was nominally set to be 5, but large (10) and small (1) values were also used. Similarly, the number of nodes was nominally set to be 2, but large (10) and small (1) values were also used.

\( T_{rnd} \) (180 tests): cubic and quartic functions were integrated (as above). Integration limits were randomly selected from \([-10, 10]\). The numbers of panels and nodes were selected by randomly choosing integers from \([1, 10]\).

**GEPiv**

**Inputs**: (1) a coefficient matrix and (2) a corresponding right-hand side vector. If the system to be solved is given by \( Ax = b \) then the first input is \( A \) and the second input is \( b \).

**Output**: a vector representing the solution to the system of equations. That is, \( x \) such that \( Ax = b \) is satisfied.

**Constraints**: all inputs must be real and the system must have a solution; if a pivot is encountered that is less than 50\(\epsilon\) or the dimensions of the inputs are incompatible then it will be assumed that there is no solution. Contents of output vector must be real.

\( T_{Pop} \) (4 tests): For the first three tests, the coefficient matrix of the \( n \)th test was selected to be an \( n \)-by-\( n \) matrix. The two inputs for the first test were randomly selected from \([0, 100]\). The coefficient matrices for the two other tests were randomly selected using the requirement that the system to be solved would have a condition number greater than \(10^{20}\) — the two matrices that resulted had condition numbers of \(10^{25}\) and \(10^{105}\). The other inputs were selected so that they contained both a positive
and negative. The second input for the third test also contained a 0. The fourth test is a simple test that was added after it was found, by inspection, that four unrevealed terminal mutants could be revealed by a specific test: a two-by-two identity matrix was used as the first input, and the vector \([1; -1]\) was used as the second input.

**T_{rmd} (40 tests):** The size of each coefficient matrix was randomly selected from the integers in [1, 10] while the values in each matrix were randomly selected from [0, 10^{105}]. The values in the right-hand side vector were randomly selected from [-100, 100].

**nwtsqrt**

**Inputs:** (1) non-negative real number \(x\) and (2) initialization value for the Newton method iterations.

**Output:** approximate value of \(\sqrt{x}\).

**Constraints:** iterations must converge.

**T_{Pop} (5 tests):** for the first input, 50 was used as a nominal input, and 0 and 100 were used as small and large inputs. For the second input, 10 was used as a nominal input, and \(\epsilon\) and 100 were used as small and large inputs.

**T_{rmd} (50 tests):** Both inputs were randomly selected from (0, 100).

**odeRK4**

**Inputs:** (1) a function that evaluates the right-hand side of an ODE that is written in standard form, (2) the stopping value of the independent variable, (3) step size for advancing the independent variable, and (4) initial condition of the dependent variable.
CHAPTER 5. MST EXPERIMENTS

**Output**: a vector containing the numerical solution at each step of the independent variable.

**Constraints**: the input must be defined at every step of the independent variable. The output vector must contain only real numbers. It is not strictly necessary that the step size be smaller than half the stopping point, but if this condition is violated the outputs that are produced are trivial.

**$T_{Pop}$ (24 tests)**: three Ordinary Differential Equations (ODEs) were solved:

\[ y'(t) = t - y(t) \]
\[ y'(t) = y(t)/(t + 1) \]

and

\[ y'(t) = t/(y(t) + 1) \]

For the second input, the values used were: 10 (nominal), 0 (small), and 100 (large).

For the third input the values used were: 1 (nominal), 0.1 (small), and 100 (large).

For the fourth input values used were 10 (nominal), $\epsilon$ (small), and 100 (large). Finally, a special test using the inputs 1e-50, 1e-51, and 1e-50 was conducted once for each ODE.

**$T_{rnd}$ (240 tests)**: The ODEs used for $T_{Pop}$ were also used for $T_{rnd}$. The second and fourth inputs were randomly selected from $[\epsilon, 100]$ while the third input was randomly selected from $[\epsilon, 10]$.

**powerit**

**Inputs**: (1) a matrix and (2) the number of iterations to conduct.

**Output**: approximate value of the largest eigenvalue of the input matrix.
Constraints: iterations must converge.

\( T_{Pop} \) (6 tests): the two input matrices that were used were the latter two coefficient matrices of the \texttt{GEPiv} Popperian tests; that is, they were matrices with large condition numbers of \( 10^{25} \) and \( 10^{105} \). Number of iterations was set to 1, 10, and 100. All six possible combinations of these inputs were tested.

\( T_{rnd} \) (60 tests): the size of each coefficient matrix was randomly selected from the integers in \([1, 10]\) while the values in each matrix were randomly selected from \([0, 10^{105}]\). The number of iterations was randomly chosen from the integers in \([1, 100]\).

\texttt{simpson}

Inputs: (1) the function to be integrated, (2) the upper bound of integration, (3) the lower bound of integration, and (4) the number of panels used for integration.

Output: approximate value of the integral.

Constraints: the function must be defined at any point between and including the integration bounds, the integration bounds must be real, and the number of panels must be a positive integer. Output must be scalar and real.

\( T_{Pop} \) (18 tests) and \( T_{rnd} \) (180 tests): the tests of this function are identical to the \texttt{gaussQuad} tests except that the number of nodes argument has been removed from each test.

\texttt{sphereFnet}

Inputs: (1) velocity of sphere, (2) mass of sphere, (3) diameter of sphere, (4) value of gravity, (5) density of the fluid in which the sphere is falling, and (6) viscosity of the fluid in which the sphere is falling.
**Output**: the net force on the sphere.

**Constraints**: all inputs are real scalars. The output must be real.

$T_{Pop}$ (18 tests): execution through this function can choose one of four branches depending on the value of the Reynold’s number that is calculated at the start of the program. Therefore, the first six Popperian tests were selected in an attempt to ensure that all paths were traversed. However, it was noted that these tests weakly exercised code that involved the last input argument. In an attempt to address this shortcoming, the first argument from each of the six initial tests was then used again with all the other arguments, except the last one, set to 1000—the last argument was set to $10^6$. These six test were repeated except that the last argument was set to 1. Finally, a test was added to reveal an unrevealed terminal mutant: velocity was set to $\epsilon$ and all other inputs were set to one.

$T_{rnd}$ (180 tests): The first and sixth inputs were randomly selected from $[\epsilon, 10^6]$ while the second to fifth inputs were randomly selected from $[1, 1000]$.

### 5.4 Analysis of the Test Outputs

#### 5.4.1 Analysis Procedure

Each target function was mutated using the Python portion of MATmute. All tests for each target were run on the original version of the target and its mutants by the `matmute` function. The results from these runs were sent by `matmute` to `get_error` to determine the value of $\gamma$ for each test-mutant pair. Error results for each target function were stored in an error matrix in which row $i$ was used to store the results for the $i$th mutant and column $j$ was used to store the results for the $j$th test. Therefore,
position \((i, j)\) in the error matrix for some target would store the \(\gamma\) score of mutant \(i\) of that target when it was run using test \(j\). Error matrices for \(T_{cmb}\) were formed for each target by horizontally concatenating the error matrices that were formed using \(T_{Pop}\) and \(T_{rnd}\).

Unrevealed mutants were identified by looking for rows in these error matrices that contained only zeros. 55 unrevealed mutants were discovered. Manual inspection revealed that 50 of these mutants were practically equivalent\(^3\) while it was determined that the other five could be revealed by adding two tests—one test for \(\text{GEPiv}\) revealed four of these and one for \(\text{sphereFnet}\) revealed the other one. These two tests were added to the Popperian set and 20 pseudo-random tests were added to the pseudo-random sets to maintain the ten-to-one size ratio for the two sets. All mutants of \(\text{sphereFnet}\) and \(\text{GEPiv}\) were then retested.

“Clean” versions of all error matrices were formed by removing mutant rows that were not viable. That is, if every element in row \(i\) of a \(T_{cmb}\) error matrix for some target was equal to infinity, then row \(i\) would be removed from all error matrices for that target. The rows of the 50 mutants that were found to be equivalent (as detailed above) were also removed. These clean error matrices were used for all analysis unless otherwise specified.

\(\Gamma\) scores for each mutant were derived from the error matrices by taking the maximum value from each row of every matrix and then storing these values in lists for analysis. These lists were used to determine detection counts for each test set by counting the number of \(\Gamma\) values that were greater than or equal to \(\gamma_d\) at every sampled \(\gamma_d\) point. These sampling points were also derived from the \(\Gamma\) score lists by \(^3\)Making this judgement requires that the inspector has a strong understanding of each mutation’s effects—it is possible that some non-equivalent mutants were mistakenly identified as practically equivalent.
finding all the unique \( \Gamma \) values in each list.

Minimal sets of equivalent tests were found for each test set by, first, collecting lists of the mutants that were maximally revealed by each test. The problem of determining a minimal set of equivalent tests can then be understood as a set cover problem: the set of all revealed mutants forms a universe that is to be covered by some combination of the sets of maximally revealed mutants. Furthermore, it is a minimal set cover problem because the desired solution uses the smallest selection of sets that cover the universe. Therefore, to find the minimal sets of equivalent tests, the list of revealed mutants as well as the lists of maximally revealed mutants for each test were passed to \texttt{getminsetcover} to find the minimal set of equivalent tests. Note that these steps are all implemented in the \texttt{getkeytests} function.

Key detectors were found—by the \texttt{getkeykillers} function—using the same process as \texttt{getkeytests} except that lists of mutants that were detected by each test were passed to \texttt{getminsetcover} instead of lists of maximally revealed mutants. Key detectors were found for all \( \gamma_d \) values in the \( \gamma_d \) sampling lists discussed above. Detection counts for each key detector were found as follows:

1. A list of detected mutants \( P_D = D(\gamma_d, P_M, T) \) was formed.
2. The key detector \( k \) that detected the most mutants in \( P_D \) was found.
3. The number of mutants in \( P_D \) that were detected by \( k \) was saved.
4. Mutants detected by \( k \) were removed from \( P_D \).
5. Steps 2 to 5 were repeated until \( P_D \) was empty.
CHAPTER 5. MST EXPERIMENTS

Table 5.1: Counts of categorized mutants.

<table>
<thead>
<tr>
<th>Target</th>
<th>Non-Viable</th>
<th>Unrevealed</th>
<th>Revealed</th>
<th>Terminal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>17</td>
<td>5</td>
<td>13</td>
<td>47</td>
<td>82</td>
</tr>
<tr>
<td>gaussQuad</td>
<td>60</td>
<td>4</td>
<td>120</td>
<td>82</td>
<td>266</td>
</tr>
<tr>
<td>GEPIv</td>
<td>31</td>
<td>17</td>
<td>39</td>
<td>138</td>
<td>225</td>
</tr>
<tr>
<td>nwtsqrt</td>
<td>15</td>
<td>2</td>
<td>6</td>
<td>42</td>
<td>65</td>
</tr>
<tr>
<td>odeRK4</td>
<td>61</td>
<td>1</td>
<td>169</td>
<td>193</td>
<td>424</td>
</tr>
<tr>
<td>powerit</td>
<td>15</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>simpson</td>
<td>36</td>
<td>12</td>
<td>106</td>
<td>27</td>
<td>181</td>
</tr>
<tr>
<td>sphereFnet</td>
<td>13</td>
<td>7</td>
<td>151</td>
<td>46</td>
<td>217</td>
</tr>
<tr>
<td>Total</td>
<td>248</td>
<td>50</td>
<td>613</td>
<td>581</td>
<td>1492</td>
</tr>
</tbody>
</table>

5.4.2 Tables and Graphs of the Results

Tables and graphs that display important information are provided below. Important features of these tables and graphs are highlighted.

Tables of Mutant and Detector Counts

Table 5.1 shows the number of mutants from each target that were non-viable, unrevealed, revealed (that is, were revealed but did not terminally fail on any test), and terminally revealed (that is, terminally failed for at least one—but not all—tests) when executed using $T_{cmb}$. Note that the total number of mutants includes the non-viable and equivalent mutants.

For each mutation target, Table 5.2 shows the number of tests in the minimally equivalent set of tests $T_Q$ from $T_{cmb}$ and, at four different $\gamma_d$ values, the number of key detectors $T_K$ in $T_{cmb}$. 
CHAPTER 5. MST EXPERIMENTS

| Target      | $|T_{cmb}|$ | $|T_Q|$ | $|T_K|$ at $\gamma_d$ of 
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>0.1 1.0 10.0 Inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>binSearch</td>
<td>121</td>
<td>6</td>
<td>3 4 4 4 4</td>
</tr>
<tr>
<td>gaussQuad</td>
<td>198</td>
<td>16</td>
<td>2 3 5 2</td>
</tr>
<tr>
<td>GEPiv</td>
<td>44</td>
<td>12</td>
<td>5 5 6 5</td>
</tr>
<tr>
<td>nwtsqrt</td>
<td>55</td>
<td>3</td>
<td>2 2 2 2</td>
</tr>
<tr>
<td>odeRK4</td>
<td>264</td>
<td>21</td>
<td>2 5 8 5</td>
</tr>
<tr>
<td>powerit</td>
<td>66</td>
<td>3</td>
<td>2 2 2 2</td>
</tr>
<tr>
<td>simpson</td>
<td>198</td>
<td>16</td>
<td>2 4 4 3</td>
</tr>
<tr>
<td>sphereFnet</td>
<td>209</td>
<td>19</td>
<td>5 8 6 7</td>
</tr>
<tr>
<td>Total</td>
<td>1155</td>
<td>96</td>
<td>23 33 37 30</td>
</tr>
</tbody>
</table>

Table 5.2: Counts of minimally equivalent tests and key detectors.

$\Gamma$ Histograms

The histogram given in Figure 5.1 provides a full, coarse view of the $\Gamma$ score distribution exhibited by the mutants when they are run using $T_{cmb}$. A finer, zoomed-in view of the mutants’ $\Gamma$ score distribution for $T_{cmb}$ can be seen in Figure 5.2. These histograms were constructed by dividing each domain of log$_{10}(\Gamma)$ values into 150 bins of equal size. The number of $\Gamma$ values in each bin were then counted and divided by the total number of viable, non-equivalent mutants to determine the percentage of mutants that fall into each bin.

Important features to note in these graphs:

- The $\Gamma$ scores span a wide range: the smallest non-zero score is $\sim 10^{-15}$ while the largest non-infinite score is $\sim 10^{292}$.

- The mutants’ $\Gamma$ scores appear to be grouped in clusters, the largest of these is clearly centered near $\Gamma = 1$ (that is, near 0 in the graphs). The effects of these clusters will be observed in later analysis.
Figure 5.1: Maximum observed error ($\Gamma$) distribution for $T_{cmb}$ (full domain).
Figure 5.2: Maximum observed error (T) distribution for $T_{cmb}$ (zoomed-in).
Plots of Detection Scores

The graphs in Figures 5.3, 5.4, 5.5, and 5.6 show the detection scores for the three sets of tests as $\gamma_d$ is varied. The accumulated (larger) graphs are generated using the $\Gamma$ scores accumulated from all the target functions: 5.3 shows the full domain of exhibited errors while 5.4 provides a zoomed-in view. The target (smaller) graphs allow the reader to examine the detection behaviour of the test sets for individual target functions. Note that the axis labels are not included in the targeted plots, but they are the same as those used in the accumulated plots.

Important features to note in these graphs:
Figure 5.4: Detection scores accumulated from all targets (zoomed-in).
Figure 5.5: Detection scores for each target as a function of detection boundary $\gamma_d$ (first set).
Figure 5.6: Detection scores for each target as a function of detection boundary $\gamma_d$ (second set).
• All of the graphs exhibit a sharp drop in detection scores as $\gamma_d$ is increased past a value of 1. This drop corresponds to the cluster of $\Gamma$ scores near $\Gamma = 1$ in the histogram plots. Other smaller, but still sharp, drops in score can be seen at other points: these smaller drops correspond to the smaller clusters that were observed in the histogram of $\Gamma$ values.

• The $T_{cmb}$ test set always detects at least as many mutants as $T_{Pop}$ and $T_{rnd}$. This is expected—a test set will always be at least as strong as any of its subsets\(^4\)—and, therefore, provides some confirmation that the analysis is valid.

• The $T_{Pop}$ and $T_{rnd}$ test sets are shown to be complementary at most points as evidenced by the fact that the $T_{cmb}$ plots are distinct from the other two plots at most points—in the overview graphs the three test sets are distinct at all points. Note, however, that some of the target graphs exhibit sections where the $T_{cmb}$ results are collinear with other results; this implies that all key tests for those targets in those $\gamma_d$ domains are provided by the test set that is collinear with $T_{cmb}$.

Plots of Detection Score Differences

The graphs in Figures 5.7, 5.8, 5.9, and 5.10 can be used to compare the detection scores of the $T_{Pop}$ and $T_{rnd}$ test sets. These graphs are generated by taking the difference between the detection score of the given test set and the $T_{cmb}$ test set at all sampled points. A smaller difference in detection score implies that a test set’s strength is closer to that of the $T_{cmb}$ test set—that is, the test set with a lower difference in detection score at some point is better at that point. If the difference

\(^4\)When a test set is augmented with one or more tests it cannot result in a weaker test set.
Figure 5.7: Detection score differences accumulated from all targets (full domain).

detection score is 0 at a point then it means that the test set’s performance at that point is identical to $T_{cmb}$. $T_{pop}$ and $T_{rnd}$ are complementary at any points where both test sets exhibit nonzero differences in score.

There are four plots showing the same domains as the detection score plots above. Important features to note in these graphs:

- In Figure 5.7 we see that $T_{pop}$ appears to be closer to $T_{cmb}$—and, therefore, better—over most of the domain of $\gamma_d$ values: this is obviously true for $\gamma_d$ values greater than about $10^3$. However, it should be noted that neither test set exhibits a difference in score of 0 at any point; this implies that the two test
Figure 5.8: Detection score differences accumulated from all targets (zoomed-in).
Figure 5.9: Detection score differences for each target as a function of detection boundary $\gamma_d$ (first set).
Figure 5.10: Detection score differences for each target as a function of detection boundary $\gamma_d$ (second set).
sets are complementary over the whole domain.

- From Figure 5.8 we see that $T_{\text{rnd}}$ is closer to $T_{\text{cmb}}$—and, therefore, better—for values of $\gamma_d$ between (approximately) 0.4 and 30. However, as above, it should be noted that the two test sets are complementary over this domain.

- For the \texttt{nwtsqrt} and \texttt{powerit} functions, $T_{\text{Pop}}$ exhibits a difference in percentage of 0 over the whole domain. This implies that the $T_{\text{rnd}}$ tests are fully extraneous for these functions. This is also nearly true for the \texttt{binSearch} function, but, over most of the domain, $T_{\text{rnd}}$ detects a small percentage of the mutants (about 2%) that are not detected by $T_{\text{Pop}}$.

- The $T_{\text{rnd}}$ set appears to be better at detecting mutations in the functions that use numerical integrations—\texttt{odeRK4}, \texttt{gaussQuad}, and \texttt{simpson}—for $\gamma_d$ values that are less than, at least, 30. It cannot be seen in these graphs, but as $\gamma_d$ is increased towards infinity the two sets’ differences in scores for these three targets converge to similar values (the difference between the two sets is less than 2% for $\gamma_d$ larger than $10^{130}$).

**Test Efficiency Plots**

The graph in Figure 5.11 shows detection scores for various numbers of applied key detectors. That is, this plot gives the detection score of each test set when only the best $n$ tests from that set are used (where $n$ indicates the values on the horizontal axis). This data is provided at four $\gamma_d$ values to demonstrate how the results change as failure detection becomes more difficult. To generate the plots, the key detectors for each test set were, first, sorted by their detection counts. The sum of the detection
counts for the first $n$ tests in this sorted list was then divided by the number of mutants and plotted at the $n$th position in the graph.

The efficiency of the best $n$ tests from a set can be calculated by multiplying the corresponding detection score by the number of viable, non-equivalent mutants (1194) and then dividing the result by $n$.

Important features to note in this graph:

- For each test set and any sampled value of $\gamma_d$ (not just those shown in the graph), 11 or fewer tests are sufficient to detect 80% of all the mutants that are detected using the full test set. For some $\gamma_d$ values, as few as 5 tests are needed.

Figure 5.11: Key detector efficiency.
to detect 80% of the mutants that are detected by the corresponding full test set.

- At $\gamma_d = 0.1$, 8 tests detect more mutants than 25 tests at $\gamma_d = 1$. Similar results are apparent for many other pairs of $\gamma_d$ values.

- Test efficiency is largely independent of the set of tests that is being examined. There are small variances among test sets—these are most visible at $\gamma_d = 10$—but the point of diminishing returns appears to occur after 4 or less tests at any given value of $\gamma_d$. This observation can be mathematically formalized by noting that the second difference of the discrete data sets—a discretized approximation of the second derivative—takes on its maximum absolute value when using 4 or less tests for any sampled $\gamma_d$ value.

5.5 Discussion of the Results

5.5.1 The Mutations

Table 5.1 and Figures 5.1 and 5.2 demonstrate that the mutation operators were able to produce mutations of widely varying severity. The author did not attempt to discern correlations between specific syntactic changes and the severity of the errors that they introduce, but such work may be profitable.

Equivalent Mutants

By manual inspection it was determined that the 50 mutants that were not revealed by $T_{cmb}$ were practically equivalent. Therefore, from the perspective of traditional
mutation testing, $T_{cmb}$ could be considered to be a practically perfect test set—it kills all non-equivalent mutants of concern.

Mutant detection, however, is much more difficult than mutant killing. This is apparent from an inspection of Figure 5.4 where we see that $T_{cmb}$ detects less mutants as $\gamma_d$ is increased.

To roughly estimate how many test scores could have led to false kills—that is, kills resulting from what would probably be considered to be negligible errors—the number of $\gamma$ results that were less than $10^{-15}$ but greater than 0 were counted: 1818 test scores (0.8% of the test scores) fell within this range.\(^5\) Testers who apply traditional mutation testing must be made aware of the danger of false killing.

**Γ Clusters and Tolerable Mutants**

As noted in the discussion of Figures 5.1 and 5.2, Γ results appear to occur in clusters. The underlying cause of this clustering is unknown, but repeated error results are not the result of syntactically identically mutants: MATmute has been designed so that the production of duplicate mutants is avoided.

It is, however, possible that some mutations are functional duplicates of other mutations, that is, they contain different code but produce identical outputs. These functional duplicates may account for some of the clustering, but it would be very surprising to find that enough of these functionally equivalent mutants exist to produce the large cluster of mutants observed at $\Gamma = 1$. Another explanation is needed.

One possibility is that the Γ clusters result from mutants that all cease to be tolerable at the same value of $\gamma_{MAX}$. That is, they are not functionally equivalent,

\(^5\) $10^{-15}$ was selected because it is one order of magnitude larger than machine epsilon when using double precision numbers.
but they all exhibit the same worst case error—extensive testing may have forced them all to exhibit this same maximum error. No attempts were made to validate or falsify this hypothesis, but it should be noted that if this hypothesis is valid it would imply that the mutants in a cluster are tolerable provided that $\gamma_{MAX}$ is greater than or equal to the $\Gamma$ value where they have clustered. Researchers of traditional mutation have explored techniques that allow testers to identify some equivalent mutants (see, for example, [26]); analogous research might allow testers to identify some mutants that are tolerable for a given value of $\gamma_{MAX}$. More research is needed.

**Terminal Mutants**

51.3% of the viable, non-equivalent mutants are non-terminally revealed. It is unknown how many of these could be forced to fail terminally, but examination of the mutants reveals that some of them—perhaps most of them—will never fail terminally. By introducing safety checks into these codes—for example, by using assertions—it may be possible to make many more of these mutations terminal. MST could be used to investigate how computational scientists could use assertions and defensive programming techniques more effectively.

### 5.5.2 The Mutation Targets

Examining Figures 5.5 and 5.6 we see that `gaussQuad`, `powerit`, `sphereFnet`, and `simpson` are most weakly revealed by $T_{cmb}$: for $\gamma_d = 10^3$ the detection scores for these targets are below 56%. This suggests either that these functions are more difficult to test, or that the test selection process used to produce $T_{cmb}$ was not as effective for these functions.
On the other hand, for $\gamma_d = 10^3$, detection scores for binSearch, GEPiv, odeRK4, and nwtsqrt are above 78% suggesting that these functions are easier to test, or that the $T_{cmb}$ test selection process was more effective for these functions.

Comparing the Results from the Targets

Because different mutation targets are more or less susceptible to different mutation operators, it can be difficult to conduct valid quantitative comparisons of the results from the different targets. A target that appears to be easier to test may just be less susceptible to mutations that are hard to detect. A qualitative examination of Figures 5.5 and 5.6 illustrates how these mutation susceptibility differences can manifest themselves.

One can see that the mutants from target functions that contain a relatively high number of arithmetic statements—odeRK4, gaussQuad, sphereFnet, and simpson—appear to exhibit less drastic detection score changes—that is, smaller steps—than those functions that are largely comprised of non-arithmetic statements—binSearch, nwtsqrt, and powerit.

GEPiv is a hybrid of these two groups: the statements concerned with pivoting rely on non-arithmetic data structure manipulations, but the elimination and back substitution statements depend on arithmetic operators. Interestingly, in GEPiv’s detection score graph one can see that GEPiv exhibits behaviour consistent with both groups: large steps are seen towards the left side of the graph and small steps are seen towards the right side of the graph.
Target Complexity

From Table 5.2 it appears that the number of minimally equivalent tests and the size of the key detector sets is impacted by the complexity of the code that is being mutated. That is, the simplest functions—\texttt{binSearch}, \texttt{nwtsqrt}, and \texttt{powerit}—require the smallest numbers of tests.

More detailed studies might be able to find a correlation between complexity metrics and the size of key detector or minimally equivalent test sets.

5.5.3 The Tests

Most of the author’s analysis efforts have focused on test set assessments and comparisons. Attempts will be made to draw some lessons from the results.

\textbf{\textit{T}}_{\textit{Pop}} \text{ versus } \textbf{\textit{T}}_{\textit{rnd}}

As can be seen in Figures 5.7, 5.8, 5.9, and 5.10 and as was highlighted in Section 5.4, \textit{T}_{\text{Pop}} and \textit{T}_{\text{rnd}} are complementary test sets. Scientific software testers should be careful to select both nominal and boundary pushing tests when they are scrutinizing scientific software.

Having said that, it should be noted that \textit{T}_{\text{Pop}} detects more mutants than \textit{T}_{\text{rnd}} at many points, and that \textit{T}_{\text{Pop}} does this using one tenth the number of tests used by \textit{T}_{\text{rnd}}.

To conduct a fair comparison between the two test selection techniques, subsets of \textit{T}_{\text{rnd}} that were the same size as \textit{T}_{\text{Pop}} were selected and evaluated for each target. These subsets of \textit{T}_{\text{rnd}} exhibited inconsistent strength and were often weaker than the full \textit{T}_{\text{rnd}} set. Figure 5.12 shows what Figure 5.8 looks like when one of these smaller
sets of pseudo-random tests is used.

$T_{Pop}$ and $T_{rnd}$ both reveal mutants that were not revealed by the other set: $T_{Pop}$ reveals 24 mutants that $T_{rnd}$ did not reveal while $T_{rnd}$ reveals 5 mutants that $T_{Pop}$ did not reveal.

Also, $T_{Pop}$ and $T_{rnd}$ both terminate mutants that were not terminated by the other set: $T_{Pop}$ terminates 48 mutants that $T_{rnd}$ did not terminate while $T_{rnd}$ terminates 14 mutants that $T_{Pop}$ did not terminate. 20 of the mutants that $T_{Pop}$ terminated were not revealed by $T_{rnd}$ while $T_{Pop}$ reveals all mutants that are terminated by $T_{rnd}$.

Again, these observations imply that the two sets are complementary but that $T_{Pop}$ is stronger. Effective scientific software code scrutinization tests must push the boundaries of the software.
5.5.4 Test Efficiency

Figure 5.11, arguably, shows the results that are most relevant to computational scientists. In particular, two important conclusions can be drawn from the data in the figure. First, detection boundaries strongly affect the ability of tests sets—large or small—to detect code faults. Second, by choosing a few good tests it may be possible to detect most of the faults in a scientific code.

Detection Bounds Strongly Influence Testability

Figures 5.4 and 5.11 demonstrate that mutations become much more difficult to detect as $\gamma_d$ is increased. In particular, Figure 5.4 shows that detection scores plummet as $\gamma_d$ is increased past, approximately, 0.8. In Figure 5.11 we note that increasing $\gamma_d$ causes test set detection scores to plateau sooner and at lower scores. These observations imply that, in some situations, it is more important to tighten bounds on acknowledged errors and reduce the uncertainty in available oracles than to conduct more tests.

Better test sets would reduce the impact of raising $\gamma_d$, but tolerable mutants cannot be forced to exhibit error that is greater than some finite bound. In other words, these tolerable mutants cannot be detected at some values of $\gamma_d$. (It is not known how many of the studied mutants are tolerable for any $\gamma_{MAX}$, but two tolerable mutants were identified by inspection and it is very likely that more exist.) Furthermore, while it is likely that stronger tests do exist, the author would not know how to conduct a fruitful search for these tests—traditional test selection techniques have not been assessed using sensitivity studies. Note that from Table 5.2 we know that only 96 of the 1155 tests in $T_{cmb}$ are actually used, and only about a third of the 96 tests are
needed to detect all detected tests at any given value of $\gamma_d$: the author questions the effectiveness of adding more tests.

These observations strongly indicate the importance of acknowledging the oracle and tolerance problems. Here the MMS has already set a strong example: by working backwards from the solution and by checking solution convergence rates, the MMS minimizes oracle uncertainty and error tolerance and dramatically improves effectiveness. Conversely, these results demonstrate that it is extremely weak to declare that an output “looks about right.” In particular, the effectiveness of code testing quickly drops as $\gamma_d$ is increased past approximately 0.8. An oracle that can only judge accuracy to within an order of magnitude is of questionable value for the purpose of fault detection via testing.

A Few Good Tests

Provided that $\gamma_d$ can be kept small (as discussed above), the curves in Figure 5.11 suggest that a few well selected tests can detect most code mutations in the eight mutation targets.

The existence of the plateaus observed in Figure 5.11 is not surprising: one would expect that the addition of new tests would have little effect once most of the obvious mutations had been detected. What is surprising, however, is the fact that so many of the mutations are so easily detected. The author had expected that more tests would have to be applied before the plateaus were reached. The efficiency of a small number of tests is impressive.

Provided that testers are able to select tests that are comparably efficacious to those studied here, these results suggest that scientific software code quality would
strongly benefit from a modicum of well-directed testing effort. It would also be interesting to investigate the mutation detection abilities of verification and validation tests.

These observations suggest that, like scientific falsification, code falsification does not require that a tester check all possible predictions. Instead, it appears as though a majority of problematic faults can be found by checking a few novel or key predictions. If a scientific theory produces accurate novel predictions it does not prove that the theory is correct, but it does allow scientists to place more trust in the theory. Similarly, if a program produces accurate outputs when run using some key tests it does not prove that a code is correct, but it does suggest that a code is deserving of some trust.

It should also be noted that more complex codes, like more complex theories, seem to require more tests. The claim that a small number of tests are sufficient to falsify a code does not mean that a constant number of tests are sufficient: Table 5.2 suggests that more tests are needed to test more complex codes.

### 5.5.5 Cautions

These early results are promising, but some cautions and limitations must be noted.

First, some observations in this thesis may be unique to small code units of the type that were tested—the results may not be applicable to larger codes. Larger scientific codes are formed by collecting small code units of the type that were tested here, and, therefore, it is hypothesized that many of the conclusions of this thesis can be scaled up for application to larger codes. Studies of larger codes that combine numerical routines with other accessories (for example, input/output calls or complex
data structures) are needed to confirm this hypothesis.

Second, in the context of sensitivity testing, the mutation operators that were used may be poorly representative of the types of code faults that frequently occur in scientific codes. Other mutation operators might result in mutants that behave differently and are harder or easier to detect. Studies using other sets of mutation operators would provide valuable reference data for comparison, and MST mutation operator representativeness research might provide better guidelines for mutation operator selection.

Third, it may be that the effective tests are actually quite difficult to select \textit{a priori}: recall that the small sets of effective tests that were selected in this work were selected after the fact from much larger sets of tests. More research is needed to determine how to best choose effective scientific software tests: an examination of the common features of key tests may lead to helpful discoveries.
Chapter 6

Conclusion

MST is a product of fruitful cross-disciplinary research focused on the problem of code faults in scientific software. After identifying the unique challenges faced by scientific software testers, it was possible to modify a software engineering technique to better suit the needs of scientists. It is believed that the result—MST—is of significant interest to both software engineers and scientists.

This thesis presents early results—much more work is needed—but it is already apparent that MST encourages shifts in the ways that software engineers and computational scientists interact with their conceptions of correctness: software engineers may discover new ways to study code faults when they relax their focus on strict output correctness while scientists may find that putting some focus on code correctness will help them increase the quality of their software.
6.1 Attention: Testing Researchers

Computational scientists need domain specific testing techniques. By ignoring the oracle and tolerance problems, software testing researchers are limiting the scope and applicability of their work, and they may even be negatively impacting the validity of the test evaluation techniques that they develop. For example, this thesis shows that when oracle uncertainties and/or output tolerances are ignored, test evaluation techniques can overestimate test set adequacy. In particular, arguments—found in Section 4.2—and experimental results—found in Section 5.5.1—have been used to show that strict equality should not be used when assessing mutants from codes that use floating point numbers.

Furthermore, the MST results in this thesis have shown that code faults can still be detected and studied when accuracy, rather than correctness, is assessed. By using variable detection bounds, testers may also be able to explore the consequences of oracle faults and errors; considering the fact that testers (human or automatic) are fallible, such research is urgently needed.

Finally, by exploring mutation sensitivity, testing research can begin to uncover possible correlations between syntactic faults and error amplitudes. Such research may even lead to new testing techniques. Studies of possible correlations between complexity measures and numbers of key detectors may also lead to interesting discoveries.

Providing scientific software testers with test selection techniques that enable them to choose good tests would allow them to consistently and effectively falsify scientific codes. If these tests detected a large number of faults, it would suggest that the code should be more carefully scrutinized. If, on the other hand, these tests did not locate
many faults then the code could be verified and/or validated with more confidence. In short, software engineers can help computational scientists improve their science.

6.2 Attention: Computational Scientists

At this time, the MST technique itself is of more use to testing researchers than to computational scientists, but many of the observations presented in this thesis are of strong relevance to computational scientists.

This thesis has argued that scientific software code quality can be significantly improved through code scrutinization activities. In particular, experimentation using MST has shown that code testing can be effectively used to falsify codes provided that certain conditions are met: namely that (1) the codes are small, (2) sufficiently accurate oracles are available, and (3) suitable test selection techniques are available.

It may be that the first of these two conditions can be dropped: the experiments in this thesis were conducted on small codes, and, therefore, this condition is included as a caution. It is hypothesized that the conclusions in this thesis are also applicable to larger codes, but more research is needed to confirm this hypothesis.

The second condition is one that must be imposed as a result of the experimental findings presented in this thesis. Results discussed in Section 5.5.4 show that test set detection abilities quickly plummet as applied detection boundaries are increased past a relative error of approximately 0.8. Oracles that are only accurate to within an order of magnitude are of questionable value.

The third condition is one that researchers should strive to address. MST can be applied by researchers who wish to begin investigating the effectiveness of test sets and testing techniques. By applying MST to existing V&V test sets, computational
scientists could begin to identify shortcomings in their testing methodologies.

6.3 Code Fault Sensitivity Testing in The Future

In the process of developing and defining the fundamentals of MST, this thesis has defined a number of concepts and terms that are related to code fault sensitivity testing. Other researchers are encouraged to adopt, improve, and augment these definitions in their own research efforts. Much more code fault sensitivity research is needed.

For example, research built on the concepts of detection boundaries and tolerable mutants/faults would be of great use to practitioners who must assess the quality of numerical and scientific codes. Oberkampf et al. write, “There are no straightforward methods for estimating, bounding, or ordering the contributions of unacknowledged errors.”[23] Perhaps tolerability research could lead to discoveries that would allow scientists to put tentative bounds on the unacknowledged errors that stem from faults in their codes? Theoretical and experimental investigations of these problems would be of great value.

In short, this thesis lays the groundwork for code fault sensitivity research, it is hoped that others will begin to build on it.
Bibliography


Appendix A

MATmute Code Listings

For reasons outlined in section 5.1.2, the MATmute code listings are divided into Python modules and MATLAB functions. The Python modules are used to generate mutants of MATLAB code while the MATLAB functions are used to execute and analyze these mutants.

Note that the code included here is identical to the code which produced the results in the thesis except that some statements have been split using “...” and some code comments have been abbreviated or split so that the text would not spill over the edges of the page.

All documentation for these codes is internal—that is, in the comments. See \_init\_\_py\_ in section A.1.1 for an overview of the role played by each Python module, and see the header of matmute.m in section A.2.6 for a description of how MATmute is used in MATLAB. The source code for the latest version of MATmute is available at matmute.sourceforge.net.
A.1 Python Code for Mutant Generation

A.1.1 __init__.py

The mute package contains modules which are used to mutate program code for testing purposes.

Currently, only MATmute (for mutating MATLAB code) has been implemented, but parts of the package (including the structure and some of the functions) can be used as a starting point to construct mutation systems for other programming languages.

The following is an outline of the modules in their current form:

- matmute: This is intended to be the "__main__" module for the MATmute system. It takes the target and execution options from the command-line, creates the appropriate objects, and instructs those objects to output the mutants in the way that was specified at execution.

- mutants: Defines the classes and methods which are responsible obtaining, storing, and outputting the mutants. Every language needs its own mutants class which outputs the mutations in the correct format.

- mutatee: Defines the classes which are responsible for obtaining and storing information about the file or function which is to be mutated. Every language needs its own mutatee class which parses the input code into a usable format.

- mutator: Contains a generalized function (mutate) which calls a given instance of operators on a given instance of mutatee in order to generate the mutants. This is called by a mutants object when it wants to obtain a set of mutants.

- operators: Defines classes of operators to be used on a given type of code. These operators are called by the mutate function to generate language specific mutants. Therefore, every language needs its own class of operators.

- ops_config: Defines configuration data for each class of operators. This is intended to allow easy reconfiguration of the behaviour of some of the operators. Every language needs its own configuration class to match its operator class.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with MATmute. If not, see <http://www.gnu.org/licenses/>.

A.1.2 matmute.py

'''
Front-end for the Python part of the MATmute mutation system.

When run as __main__ and given proper command-line input this runs the
MATmute mutation system on a MATLAB function.

Before running the MATmute system read the INSTALL.txt file to make sure that
your computer is set up correctly.

This module should be called from the command-line while in the directory that
contains the .m file of the function to be mutated. All MATmute .py files
must also be in the directory with the .m file or on the PYTHONPATH. The
command-line call sequence used to produce the mutated .m files is

    python -m matmute [name_of_function] [options]

Where [name_of_function] is the name of the .m file that is to be mutated and
[options] are optional inputs which are are listed later in this documentation.
The program will print the IDs of the mutants that were created to the standard
output. The mutant files are named using these IDs (plus .m) and are stored in
a directory named [name_of_function]_mutes.

NB: The function must be stored in an .m file with called [name_of_function].m.

EXAMPLE 1: To generate mutates of a function named "test" (in an .m file named
"test.m") execute:

    python -m matmute test

OPTIONS: The following options can be used to change the behaviour of the
mutation system when they are given on the command-line following the string
identifying the function:

    --debug (or -d) [mode]
    Alters the way in which the mutants are output. These alternate modes
can be useful for inspecting the mutants which are generated.
    mode 0: normal behavior -- this is the only mode which creates files
    mode 1: prints out mutation list for each line of code
    mode 2: prints mutants to standard out instead of making files

    --ops (or -o) "[space delimited list of mutation operators]"
    Allows the user to choose the mutation operators which are applied. By
default all operators are used, but if this flag is used then only the
space seperated operators listed after it will be applied. (Note that the
operator list should be enclosed in quotation marks.) The possible
operators are:
    crp: constant replacement
    neg: branch expression negation
    opr: operator replacement
    sdl: statement deletion

    --inferr
    Specifies the behaviour of the mutants when loops exceed the tick limit.
    If this flag is not given then loops will just be exited when the tick
    limit is used up, but the program will continue executing. When this flag
    is given then the program will generate an error and exit when the tick
EXAMPLE 2: If one wishes to mutate a function named "test" using only the statement deletion and constant replacement mutation operators then the following should be entered on the command-line:

    python -m matmute test -o "sdl crp"

NB: Any [options] arguments must be listed at the end of the execution command. The opening part of the command must always be entered as shown above. An out-of-order execution command will not function correctly.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with MATmute. If not, see <http://www.gnu.org/licenses/>.

```
import sys
from sys import argv

import mutatee
import operators
import mutants

# if run as main (the intended usage)
if __name__ == "__main__":

    # set default values for options
    debug_mode = 0  # no debugging
    op_set_tpl = None  # use ops listed as defaults
    fail_on_bad_loop = False

    # check for other command-line arguments (see the doc string of this # module for a full explanation of the options)
    for i in range(2, len(argv)):
        if argv[i] in ("-d", "--debug"):
            debug_mode = int(argv[i+1])
        elif argv[i] in ("-o", "--ops"):
            op_set_tpl = tuple(argv[i+1].split())
        elif argv[i] in ("--inferr"):
            fail_on_bad_loop = True

    # end for
```
# create instance of mutation target (holds code listing)
mutatee_obj = mutatee.MatlabMuteTarget(argv[1], fail_on_bad_loop)

# create instance of operators object (holds set of operators to apply)
operators_obj = operators.MatlabMuteOps(op_set_tpl)

# create instance of mutants (creates and holds mutants)
mutants_obj = mutants.MatlabMutes(mutatee_obj, operators_obj)

# choose behaviour based on value of debug_mode
if debug_mode == 0:
    # create mutant files and print out ID names
    print "\n".join(mutants_obj.create_mute_files())
elif debug_mode == 1:
    mutants_obj.print_mn_lsts() # print lists of mutations to console
elif debug_mode == 2:
    mutants_obj.print_mns() # print mutants to console
else:
    print "Invalid debug mode selected."
    sys.exit(1)

# end if __main__

A.1.3 mutants.py

'''Classes responsible for getting and storing mutants.

Each class should have at least one corresponding mutatee class which handles
the function which is being mutated. Methods to store mutants to disk are
also included in these classes.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

MATmute is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with MATmute. If not, see <http://www.gnu.org/licenses/>.

,,,

import os
import re
from mutator import mutate

class MatlabMutes:
    '''Class which handles mutants generated from MATLAB code.'''
def __init__(self, mutatee_obj, ops_obj):
    '''Get and store a list of mutations.

    Inputs:
    - the mutatee object which contains the code listing
    - the operators object which defines the mutation operators which are to be applied

    # keep track of mutatee that code came from
    self.__mutatee_obj = mutatee_obj

    # name of storage directory; set when mutants are stored
    self.__mn_dir = ""

    # call mutate to get the mutants
    self.__mn_lstlst = mutate(mutatee_obj, ops_obj)

    # A unique ID string is created for each mutant (for later use)
    self.__mn_id_lst = []
    cnt = 1 # counter to determine mutant number
    for i in range(1, len(self.__mn_lstlst)):
        for j in range(1, len(self.__mn_lstlst[i])):
            self.__mn_id_lst.append("Num" + str(cnt) + "Line" + str(i) + "Vers" + str(j))
            cnt += 1
    # end for
    # end for

    self.__num_mutants = len(self.__mn_id_lst)

    # end __init__ method

def __output_mutes(self, create_files = True):
    '''This is used to generate the full mutant programs.

    The mutations for each statement have already been generated but it is necessary to form the actual mutated programs from these lists of mutants. That is the purpose of this method.

    Input:
    - a boolean:
      if True the function produces the mutant program files
      if False the function prints the mutant programs to standard out

    Output:
    - the list of mutant IDs

    if create_files:
        # name the directory where the mutants will be stored
        self.__mn_dir = self.__mutatee_obj.get_fn_name() + "_mutes/"
        # make directory to store mutants (if not already existent)
        if not os.path.isdir(self.__mn_dir):
            os.mkdir(self.__mn_dir)
    else:
        print "Mutant storage directory: ", self.__mutatee_obj.get_fn_name() + "_mutes/"

    # Save an unmutated version of the function in the mutant directory
# (named "unmutated.m", it may be used for comparison purposes).
# Note that this will not be a duplicate of the original function
# because it was parsed and instrumented when mutatee was initialized.
output_str = ""
for mn_lst in self.__mn_lstlst[1:]:
    output_str += mn_lst[0] + "\n" # mn_lst[0] is unmutated

if create_files:
    file = open(os.path.join(self.__mn_dir, "unmutated.m"), "w")
    try:
        file.write(output_str)
    finally:
        file.close()
else:
    print os.path.join(self.__mn_dir, "unmutated.m") + ":"
    print output_str

# Now the files are actually created using the mutant ID strings.
# A uniquely named file for each mutation is created in the mutant
# directory.

cnt = 0 # loop counter used to track current mutant index
for i in range(1, len(self.__mn_lstlst)): # loop over statements
    for j in range(1, len(self.__mn_lstlst[i])): # over stmnt mutants
        output_str = "" # stores string to write to mutant file
        # get string of full code with only line (i) modified
        for k in range(1, len(self.__mn_lstlst)):
            if k != i: # if not at line (i) then print unmutated code
                output_str += str(self.__mn_lstlst[k][0]) + "\n"
            else: # when at line (i) then print out mutation (j)
                output_str += str(self.__mn_lstlst[k][j]) + "\n"
        # end for
        # form filename using mn_id
        filename_str = os.path.join(self.__mn_dir,
                                     self.__mn_id_lst[cnt] + ".m")

        if create_files:
            file = open(filename_str, "w")
            try:
                file.write(output_str)
            finally:
                file.close()
        else:
            print filename_str + ":"
            print output_str
            print ""

        cnt += 1 # increment position count
# end for
# end for

return self.__mn_id_lst
# end __output_mutes method

def create_mute_files(self):
    '''Creates the files which store the mutants for execution.'''
    Output:
APPENDIX A. MATMUTE CODE LISTINGS

- the mutation IDs (i.e., filename without ".m")
  
  # call output_mutes with True to create the mutation files
  return self.__output_mutes(True)

  # end create_mute_files method

  def print_mn_lsts(self):
    '''Prints the mutations for each code statement to the console.
    (This is useful for tracking and inspecting the mutations.)

    The unmutated statement is printed first, and then all the mutations
    for that statement are printed. Each mutation is preceded by a
    a 3-tuple which indicates (respectively) mutation number, line number,
    and line version number.
    '''

    cnt = 1
    for i, mn_lst in enumerate(self.__mn_lstlst[1:]):
      print "Original: " + mn_lst[0]
      for j, mn in enumerate(mn_lst[1:]):
        print "(" + str(cnt) + ", " + str(i+1) + ", " + \
          str(j+1) + "):" + mn
        cnt += 1
    print ""

    # end print_mn_lsts method

  def print_mns(self):
    '''Prints all mutations to the console instead of creating files.
    (Useful for debugging purposes.)

    Mutant file ID is printed before the full mutated program is printed
    (only one mutation should be present in each program). Blank lines
    are used to separate the mutant program listings.
    '''

    # call output_mutes with False to print mutants to the console
    self.__output_mutes(False)

    # end print_mns method

  def get_mn_id_lst(self):
    '''Returns the list of mutants IDs.
    The mutant IDs are used to form the mutant's file names.
    '''

    return self.__mn_id_lst

  # end get_mn_id_lst

  def get_num_mutants(self):
    '''Returns the number of mutations.''

    return self.__num_mutants

  # end get_num_mutants

  def get_mn_dir(self):
Returns the directory where the mutants are saved (if existent).

```python
return self.__mn_dir
```

# end MatlabMutes class

A.1.4 mutatee.py

Defines classes used to store the attributes of the mutation target.

(Mutatee is meant to indicate the subject of mutation.)

In general, the mutatee module is meant to define functions and classes which get and store information about a mutation target. This includes methods to retrieve the code and translate it into the appropriate format.

Classes should be defined for specific languages using language specific methods.

Currently the MatlabMuteTarget class is defined to deal with MATLAB functions and their corresponding .m files.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with MATmute. If not, see <http://www.gnu.org/licenses/>.

```python
import re
import sys
class MatlabMuteTarget:
    '''Used to obtain and store attributes of MATLAB functions.'''
    def __init__(self, fn_name, fail_on_bad_loop = False):
        '''Reads a function's .m file and prepares it for mutation.
        The .m file to be mutated is read in, parsed and instrumented, and then stored as a list of statements.
        Inputs:
            - name of the MATLAB function to be mutated
        NB: The name of the function's .m file must follow MATLAB conventions. I.e., the name of the .m file is constructed by appending ".m" to the name of the function.'''
```
# assign inputs as data attributes
self.__fn_name = fn_name
self.__fail_on_bad_loop = fail_on_bad_loop

# attempt to open .m file for for reading; if successful store the
# code as a string
try:
mfile = open(fn_name + "\.m", "r")
except IOError:
    print "Problem accessing .m file."
    print "Is the program being executed in the right directory?"
    print "Is the file name given by 'function_name'.m ?"
    sys.exit(1)
try:
    fn_code_str = mfile.read()
finally:
    mfile.close()

# create code listing
self.__code_lst = self.__create_code_lst(fn_code_str)

# end __init__ method
def __create_code_lst(self, code_str):
    
    Instrumentation process removes comments (except those used by the
    mutator) and line continuations, breaks up multi-statement lines, and
    inserts code to monitor loops and catch errors.

    Inputs:
    - the program code as a string

    Output:
    - a list of instrumented code statements
    
    def __is_nested_func(index):
        
        Checks if a nested function is declared at given index.

        Inputs:
        - index in code_lst of the function declaration to be checked

        Output:
        - True if nested function, False otherwise
        
        # compile reg exps for "function" and "end"
        fn_re = re.compile(r"\bffunction\b")
        end_re = re.compile(r"\bend\b")

        # if code_lst[ind] is not a function declaration then something
        # has gone wrong and the program should quit
        if fn_re.search(code_lst[index]) == None:
            print "Parsing error: check for nested function called on " + 
            "a line which does not declare a function."
            sys.exit(1)

        # the following keywords must be matched with an end statement
        keywords = ("for", "while", "switch", "try", "if")
        keyword_relst = []
        for keyword in keywords:
# compiles the reg exprs for the keywords
keyword_relst.append(re.compile(r"\b" + keyword + r"\b"))

# Traverse code_lst starting from the given index. Add 1 for
# keywords and "function"s and subtract 1 for "end"s. The only
# way that this count can possibly go below 0 is if more "end"s
# than "function"s are seen and this will only happen if the
# traversal starts at the declaration of a nested function.
nest_cnt = 0 # start counter at 0
for stmt in code_lst[index:]:
    if fn_re.search(stmt) != None:
        nest_cnt += 1
    elif end_re.search(stmt) != None:
        nest_cnt -= 1
    else:
        for keyword_re in keyword_relst:
            if keyword_re.search(stmt) != None:
                nest_cnt += 1
    # end if

    # If the count ever goes below zero then the code traversal
    # must have started at a nested function so return True.
    if nest_cnt < 0:
        return True

    # end for

# end __is_nested_func submethod
def __find_string_end(index):
    '''If index is inside a quote string then return the quote's end.

    "Quote string" is meant to indicate text that is bracketed by
    single-quotes, i.e., a string declaration in MATLAB.
    
    Inputs:
    - index in code_lst of position being checked
    
    Output:
    - integer giving the position where the quote string ends, if
      position is not in a quote string then -1 is returned
    '''

    # find the start of the current line
    line_st = code_str.rfind("\n", 0, index) + 1

    # Loop until the end of a quote string past the index is found.
    # Note that the loop will be exited (by "return") if it is
    # determined that the index is not inside a quote string.
    end = 0
    while end <= index:
        # find the start of the quote, note that a check must be
        # carried out (using a while loop) to ensure that the "'" is
        # not a transpose operator
        st = code_str.find("'", line_st, index)
        if st < 0: return -1
        while code_str[st-1].isalnum() or \
            code_str[st-1] in ("(*)", ",", ",", ",", ",."): #
            st = code_str.find("'", st+1, index)
        if st < 0: return -1
        # end while

        # find the end of the quote found above, a check for "'''
        # must be carried out to be sure the actual end is found
end = code_str.find("'', st+1)
while end != len(code_str) and code_str[end+1] == '':
  end = code_str.find(''', end+2)

# if index is not within st and end then the loop continues
# using end position to get the new start position
line_st = end + 1
# end while

# if the program reaches this point it means that a valid end has
# been found
return end

# end __find_string_end submethod

# Step 1: Remove block comments from code.
p = code_str.find("%{") # find start of block comment
while p >= 0:
  z = __find_string_end(p) # check if this is in a quote string
  if z < 0: # if not in a quote string
    q = code_str.find("%}") # find end of block comment
    # create string without comments
    code_str = code_str[0:p] + code_str[q+2:]
  else:
    p = z # if in quote string, skip to end of the string

# Step 2: Remove other comments (except commands to mutator).
p = code_str.find("%") # find start of (possible) comment
while p >= 0:
  z = __find_string_end(p) # check if this is in a quote string
  if z < 0: # if not in a quote string
    q = code_str.find("\n", p) # find end of line
    if q < 0: # if line doesn’t end in line break then reached EOF
      q = len(code_str)
    # create string without comment if not "_MUTE_" instruction
    if code_str[p:p+7] == "%_MUTE_":
      p += 7 # skip over _MUTE_ flag
    else:
      code_str = code_str[0:p] + code_str[q:]
  else:
    p = z # if in quote string, skip to end of the string

# Step 3: Remove line continuations while joining continued lines.
p = code_str.find("...") # find (possible) line continuation
while p >= 0:
  z = __find_string_end(p) # check if this is in a quote string
  if z < 0: # if not in a quote string
    q = code_str.find("\n", p) # find end of line
    if q < 0: # if line doesn’t end in line break then reached EOF
      q = len(code_str)
    # remove continuation from code (also drop excess whitespace)
    code_str = code_str[0:p] + code_str[q+1:].lstrip()
  else:
    p = z # if in quote string, skip to end of the string

# end while
# Step 4: Reformat text so that there is one and only one statement per line. Data structure declarations which span multiple lines are spliced onto one line, and multi-statement lines are divided up into single statement lines.

```python
char_lst = list(code_str)  # change to list since strings are immutable
(p, bc) = (0, 0)  # loop and bracket counters
while p < len(char_lst):
    # track brackets and parentheses
    if char_lst[p] in ("[", "(", ":"): bc += 1
    elif char_lst[p] in ("]", ")", ":"): bc -= 1
    # if multi-line data declaration replace line break with ";"
    if char_lst[p] == 
    # if statement break found then replace with line break
    elif char_lst[p] == ";" and bc == 0:
        char_lst[p] = "\n"
    # if quiet statement break found then add line break after ";"
    elif char_lst[p] == ";" and bc == 0:
        char_lst.insert(p+1, "\n")
    p += 1
# end while
code_str = ".join(char_lst)  # convert list back to string
```

# Step 5: Split text into a list where each item in the list corresponds to a statement in the code.

code_lst = code_str.splitlines()

# Step 6: Instrument loops with iteration monitoring code. This is necessary to kill infinite loops which may be created in a mutant. The number of iterations of each loop in the unmutated program is measured and can be multiplied by some preset factor. If the same loop in a mutant exceeds the result of the calculation it is killed. (Note that this functionality requires support from the MATLAB function or script which is used to run the mutants.)

```python
loop_strtpl = ("for", "while")  # these declare loops
# setup the regular expressions which are used to search for loops
loop Relst = []
for loop_str in loop_strtpl:
    loop Relst.append(re.compile(r"\b" + loop_str + r"\b"))
```

# loop until all statements have been checked for loops

```python
loop_cnt = 0  # used as an ID for each loop
while i < len(code_lst):
    # check if the statement contains any of the search expressions
    (j, match) = (0, None)  # setup loop variables
    while j < len(loop Relst) and match == None:
        match = loop Relst[j].search(code_lst[i])
        j += 1
    # if a loop is found
    if match != None:
        # form strings for later use
tick_str = "MUTE_ticks(MUTE_testcnt)"
```
ticklimit_str = "MUTE_ticklimit(MUTE_testcnt)"

# insert code in start of loop to count and monitor ticks
code_lst.insert(i+1, "%_MUTE_off")
code_lst.insert(i+2, "\t" + tick_str + " = " + tick_str + " + 1;
")
code_lst.insert(i+3, "\tif ~ isempty(MUTE_ticklimit) && " +
tick_str + " > " + ticklimit_str)

code_lst.insert(i+4, "\tend")
code_lst.insert(i+5, "%_MUTE_on")

# choose behaviour of mutant when tick limit is exceeded based
# on the value of the fail_on_bad_loop flag.
if self.__fail_on_bad_loop:
  code_lst.insert(i+4, "\terror('Tick limit exceeded. ')
else:
  code_lst.insert(i+4, "\t\twarning('mutant:" +
  "TickLimitExpired', 'Loop in mutant " +
  "exited because it exceeded its tick "+
  "limit.')")
  code_lst.insert(i+5, "\tbreak")

# NOTE: the code to declare the loop count variables as global
# is added in the next step of code processing.

# end if

i += 1 # go to next line
# end while

# Step 7: Add mutation flags around function declarations to prevent
# declarations from being mutated. Also, insert global loop counter
# declarations at the start of any functions.
fn_re = re.compile(r"bfunctionb") # reg exp to find function
i = 0 # loop counter
is_primary_func = True # first function found is the primary function
while i < len(code_lst):
  match = fn_re.search(code_lst[i])
  if match != None: # if a match is found
    # insert code to turn off mutation around function declaration
    code_lst.insert(i, "%_MUTE_off")
    code_lst.insert(i+2, "%_MUTE_on")

    # if not a nested function then declare loop monitoring globals
    # and reset ticks to 0
    if not __is_nested_func(i+1):
      code_lst.insert(i+2, "\tglobal MUTE_ticks "+
      "MUTE_ticklimit MUTE_testcnt")
      code_lst.insert(i+3, "\tMUTE_ticks(MUTE_testcnt) = 0;")

    # if the primary function then increment test count
    if is_primary_func:
      code_lst.insert(i+3, "\tMUTE_testcnt = MUTE_testcnt + 1;")
      is_primary_func = False # other functions are not primary

    i += 2 # function position has shifted down so skip two lines
else:
    i += 1 # go to next line
# end if
# end while

# Step 8: Remove blank lines and remove trailing whitespace characters
# from each line.
    code_lst = [line.rstrip() for line in code_lst
        if line != "" and not line.isspace()]

# done: return the list of code lines
    return code_lst
# end __create_code_lst method

def get_fn_name(self):
    
    # Returns name of the target function."
    return self.__fn_name
# end get_fn_name method

def get_code_lst(self):
    
    # Returns list of statements in target code."
    return self.__code_lst
# end get_code_lst method

# end MatlabMuteTarget class

A.1.5 mutator.py

'''' Defines the functions which control the mutation process.
Attempts have been made to keep the functionality very general so that any set of operators can be called on any list of input statements. This module acts as a layer between the mutatee and mutants which calls the operators. Mutatee objects store the information to which a set of operators are applied to create mutant objects. This module contains functions to coordinate that process.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with MATmute. If not, see <http://www.gnu.org/licenses/>.
''''
def mutate(mutatee_obj, operators_obj):
    """Mutates a program line-by-line using a given set of mutation operators.

    Input:
    - a mutatee object which holds listing of code to mutate
    - an instance of an operator object which defines the operators to apply

    Output
    - a list of lists (i.e, 2D list) containing all mutations of the input:
      every code statement has a corresponding inner list or "row" in the outer
      list, this inner list stores the statements mutants as they are created
      (for example, the 3rd mutation of statement 5 is at [5][3])
    """

    # Create a list to store the lists of mutants; the first entry is a
    # dummy entry which was included so that the list indexes start counting
    # from 1.
    mutant_lstlst = [['Dummy entry to offset the list indices.']]

    # loop through statements and mutate them if MUTE_off is not set
    mute_off = False
    for stmnt in mutatee_obj.get_code_lst():
        # unmutated version of statement stored as first entry of each "row"
        mutant_lst = [stmnt]

        # case statements for _MUTE_ flags
        if stmnt.find("_MUTE_off") >= 0:
            mute_off = True
        elif stmnt.find("_MUTE_on") >= 0:
            mute_off = False
        elif mute_off or stmnt.find("_MUTE_skip") >= 0:
            pass
        else:
            # Loop through the mutation operators to collect all possible
            # mutations of the current statement.
            for op_str in operators_obj.get_active_ops_tpl():
                # get operator method given in op_str
                op = getattr(operators_obj, op_str)
                # apply operator to current code statement
                mutant_lst.extend(op(stmnt))

        # Store all mutations of the current statement.
        mutant_lstlst.append(mutant_lst)
    # end for

    return mutant_lstlst
# end mutate function

A.1.6 operators.py

"""This module defines classes of mutation operators.

New operators can easily be added by defining a corresponding function. New
operators' names should be added to the default_ops_tpl variable in the
corresponding class of the ops_config module.

See descriptions in operator methods for more details of operation.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

MATmute is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with MATmute. If not, see <http://www.gnu.org/licenses/>.

'''

import re
import sys
import ops_config

class MatlabMuteOps:
    ''' Defines the mutation operators for MATLAB code.

    The operation of some of the operators is configured in the MatlabOpsConfig
class of the ops_config module.
    '''

    # Import the correct configuration variables for this class.
    __config = ops_config.MatlabOpsConfig()

    def __init__(self, input_ops_tpl = None):
        ''' Creates an operator object which applies the chosen mutators.

        Input:
        - a tuple of strings which select the mutation operators to use; each
          string item should correspond to the name of a mutation operator
          (if not given or if given None then the default set of operators
          defined in the relevant portion of ops_config will be used)
        '''

        if input_ops_tpl == None:
            self.__active_ops_tpl = self.__config.default_ops_tpl
        else:
            # check if operators named in the input exist (for security)
            for op_str in input_ops_tpl:
                if op_str not in dir(self):
                    print "Error: Non-existent mutation operator requested."
                    sys.exit(1)

            self.__active_ops_tpl = input_ops_tpl

    # end __init__ method

    def get_active_ops_tpl(self):
        #
''' Returns tuple listing this instance's active mutation operators.

This function is used by the mutate function in the mutator module in
order to determine which mutation operators should be called.

    return self.__active_ops_tpl
# end get_active_ops_tpl

def sdl(self, line): # statement deletion
    ''' Operator that deletes a statement (i.e., comments it out).

    Input:
    - a line of code

    Output:
    - a single-item list containing only the commented-out input line

    Because this operator always comments-out a line its operation is
    trivial, but it has been coded so that its interface matches the other
    mutation operators.

    # inserts MATLAB comment to mark the deletion
    return ["% statement deleted: " + line]
# end sdl operator method

def crp(self, line): # constant replacement
    ''' Operator that replaces constants (groups of digits) with new values.

    Input:
    - a line of code.

    Output:
    - a list of mutated strings where the digits of constant values are
      replaced with new values

    The value of the replacement values is controlled by the contents of
    new_val_fn in ops_default module.

    def __get_new_vals(old_val):
        ''' This function generates the replacement mutation values.
        Edit new_val_fn to change the mutations generated.

        Input:
        - the numerical value (integer or floating-point) which is to be
          changed

        Output:
        - a list of unique replacement values derived from new_val_fn

        # get list of potential values from defined function
        __computed_vals_tpl = self.__config.new_val_fn(old_val)
        new_val_list = []
        for val in __computed_vals_tpl:
if val != old_val and val not in new_val_lst:
    new_val_lst.append(val)
return new_val_lst
# end __get_new_vals subfunction

mute_lst = [] # list to store mutations

# Create regular expression which matches numbers (including floating-
# point values). RE matches simple integers, as well as numbers that
# include, start with, or end with a "." and/or which include "e"
# or "E".
num_re = re.compile(r"(?:\[0-9]*\.)?[0-9]+(?:[ eE][-+]?[0-9]+)?\.)?"
match_obj = num_re.search(line) # search for the RE from start of line
while match_obj != None: # while matches are found
    (st, end) = match_obj.span() # get start and end of match
    # get the value of the number from the digits
    num_string = line[st:end]
    # store as either float or integer depending on string contents
    if "." in num_string or "e" in num_string or "E" in num_string:
        val = float(num_string)
    else:
        val = int(num_string)
    # mutants will be created with all values in new_vals
    new_vals_lst = __get_new_vals(val)
    for new_val in new_vals_lst:
        # create mutant and append it to the list of mutations
        mute_lst.append(line[0:st] + str(new_val) + line[end:] +
            " % constant replaced: " + str(val) +
            " -> " + str(new_val))
    # end for
    match_obj = num_re.search(line, end)
# end while
return mute_lst
# end crp operator method

def neg(self, line): # negate branch expressions
    """ Operator that negates the decision in an if or while statement."
    Input:
    - a line of code.
    Output:
    - a list of mutated strings where branch conditions are negated
    """
    # the search string are set in the ops_config module
    __neg_trg_tpl = self.__config.neg_trg_tpl
    mute_lst = [] # list to store mutations
for trg in __neg_trg_tpl: # loop through the search targets
    trg_re = re.compile(r"\b" + trg + r"\b")
    match_obj = trg_re.search(line) # search for the target

    # check if the regular expression is found
    if match_obj is not None:
        # The branch expression starts at the end of the match to the
        # regular expression.
        st = match_obj.end()

        # The branch expression ends at the end of the statement, but
        # it is necessary to determine where the statement ends. If
        # it is a quiet statement then it ends with a semicolon, and
        # the end of the expression must be adjusted accordingly.
        if line[len(line)-1] == ';': end = len(line)-1
        else: end = len(line)

        # create a new string and append it to the list of mutations
        mute_lst.append(line[0:st] + " ~(" + line[st:end] + " ")" +
                        line[end:] + " % negated branch expression: " +
                        line[st:end])

    # end if

# end for

return mute_lst

# end neg operator method

def orp(self, line): # operator replacement
    '''Operator that replaces swaps code operators (e.g., + -> -, == -> ~=)

    Input:
    - a line of code

    Output:
    - a list of mutated strings where program operators are replaced with
      different operators from the same class

    To augment or modify the classes of operators which are mutated it is
    necessary to modify the "op_classes_tpl" and it may also be necessary
    to change "op_check_tpl" (both are found in the MatlabOpsConfig class
    of the ops_config module).

    Current deficiencies:
    - unary sign operators (e.g., negation by "-")) are identified as
      arithmetic operators; this can lead the creation of faulty code, but
      this is considered to be acceptable in the context of mutation
      testing because these mutants will be killed by MATLAB parse checks
      
    # See MatlabOpsConfig class for details about these variables
    __op_classes_tpl = self.__config.op_classes_tpl
    __op_check_tpls = self.__config.op_check_tpls

    mute_lst = [] # list to store mutations
for op_class in __op_classes_tpl:  # loop through the operator classes
    for op in op_class:  # loop through operators in each operator class
        # This section forms a regular expression which searches for
        # op. Care must be taken to ensure that it is not actually
        # part of a different operator. This check is done by
        # ensuring that it is not preceded or succeeded by certain
        # chars (the chars are defined in op_check_tpl when needed).
        # See definition of op_check_tpl for more details.
        (pre, post) = ("", ")
        for check_tpl in __op_checkTpl:
            if op in check_tpl[0]:
                if check_tpl[1] != ":
                    pre = "(?!\[" + check_tpl[1] + "]")"
                if check_tpl[2] != ":
                    post = "(?![" + check_tpl[2] + "]")"
        srch_str = pre + re.escape(op) + post
        srch_re = re.compile(srch_str)
        match = srch_re.search(line)  # search for the operator
        while match is not None:  # loop until no match to RE
            (st, end) = match.span()  # get position of operator
            # replace op with all substitute ops in class
            for sub in op_class:
                if sub != op:
                    # create mutate string and append it to the list
                    mute_lst.append(line[0:st] + sub + line[end:]
                    " % operator replaced: ">" + op +
                    ", ->", + sub + ":")"
            # end for
            # find the next instance of op
            match = srch_re.search(line, end)
        # end while
    # end for
# end MatlabMuteOps class

A.1.7 ops_config.py

'''This module stores operator configuration settings for easy modification.
Each class defined in this module is used by a corresponding operator class in
the operators module.'
Before modifying this file it is suggested that a backup be made so that the original configuration can be restored.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with MATmute. If not, see <http://www.gnu.org/licenses/>.

```python
class MatlabOpsConfig:
    """Configures behaviour of MatlabMuteOps operators.

The following variables are set in this class:

default_ops_tpl (a tuple of strings) -- Selects the default set of mutation operators to use (operators can also be chosen on the command-line).
For example, the tuple ("sdl", "crp") would select the sdl and crp operators (as defined in operators.py) unless a set of operators is given on the command-line (in that case the command-line inputs take precedence).

neg_trg_tpl (a tuple of strings) -- Everything in a statement which follows a string in this tuple will be logically negated.

op_classes_tpl (a tuple of tuples) -- This defines the classes of code operators (NOT mutation operators) which are considered to be "compatible" with each other. Whenever a code operator that is listed in a set is encountered in a statement, it is replaced by all the other operators in that set. Each replacement is is used to make a mutant. For example, if the set ("+", "-", "*") were given then the statement "x + y" would generate two mutants: one with "x - y" and the other with "x * y".

op_check_tpls (a tuple of a tuple and 2 strings in a tuple) -- This is a partner to the op_classes_tpl which is used to make sure that a char in a code operator is not mutated incorrectly. For example, the "<=" operator could be mutated to become "<=" if a proper check was not carried out (because the mutator would try to mutate the "<" char). In order to avoid this problem the mutator will always check if a "<" is followed by "+". If it is, then the mutator will not mutate the "<" (it will still mutate "="). The tuple defines which operators are checked and what chars are not allowed to precede or follow those operators.

new_val_fn (a function outputting a tuple of numbers) -- This is used to calculate the values that will be used to replace constants when applying the CRP operator.

```
```
# USED BY CLASSES OF THE OPERATOR MODULE
# See docstring for purpose. Any operator defined in the operators class
# can be used by including its name (a string) in this tuple.
default_ops_tpl = ('sdl', 'crp', 'neg', 'orp')

# USED BY THE NEG OPERATOR
# See docstring for purpose. Simply enter search strings in the tuple.
neg_trg_tpl = ('if', 'while')

# USED BY THE ORP OPERATOR
# See docstring for purpose. Each tuple of strings gives a set of
# "compatible" operators. To add an operator, include the search string
# which is to be found in the appropriate set (i.e., the appropriate
# tuple). To add a set, add a new tuple and fill it with operators.
op_classes_tpl = (('+', '-', '*', '/', '\', '^'), # matrix arithmetic
                  ('+', '-', '*.', './', '.\', '.^'), # element arithmetic
                  ('<', '>=', '>', '>=', '==', '~='), # relational
                  ('&', '|'), # logical
                  ('&&', '||'), # scalar, short-circuit logical
                  )

# See docstring for purpose. When detecting code operators for mutation,
# any operator in the first column which is preceded by something from the
# second column or followed by something from the third column will be
# ignored by the mutator.
# NB: The 2nd and 3rd columns take characters which are used within the
# bracketed [] part of a regular expression. Entries must be formulated
# accordingly. This means that sets of characters can be given using the
# "-" character (e.g., a-z), but if one wishes to match "-", "[", or "]"
# those 3 characters must be escaped with "\\".
# THE VALUES GIVEN HERE WILL PROBABLY NOT NEED TO BE CHANGED UNLESS
# OPERATOR MISDETECTION IS OBSERVED.
op_check_tpls = ((('+', '/', '-', '^'), '.*', ''),
                (('+', '-'), '.*', '
'),
                (('+', '-'), '.*', '
'),
                )

# USED BY THE CRP OPERATOR
# See class docstring for purpose. Simply enter new values (or equations)
# in the tuple which is returned to modify the constant values inserted
# into CRP mutations. For example the tuple (5, v*5, 0) would cause a
# constant C in a line to be replaced with 5, 5*C, and 0 in three different
# mutations.
new_val_fn = lambda self, v: (int(v==0), -v, v-1, v+1, v*0.9, v*1.1)

# end MatlabOpsConfig

A.2 MATLAB Code for Mutant Analysis

A.2.1 getapproxsetcover.m

% Uses greedy algorithm to find an approximate solution to the set cover
% problem. The size of the solution will be equal to the size of the optimal
% solution within an approximation ratio given by

APPENDIX A. MATMUTE CODE LISTINGS  124
function setcover = getapproxsetcover(universe, subsets)

% make sure that subsets can actually cover the universe
if ~all( ismember(universe, [subsets{:}]) )
    error('Subsets cannot be used to cover universe. ')
end

setcover = [];

% continue this loop until the set is covered
while ~isempty(universe)

    % determine the winner, i.e., the subset that covers the most of the
    % remaining universe
    winner = 1;
    for n = 2:length(subsets)
        if sum( ismember(universe, subsets{n}) ) > ... 
            sum( ismember(universe, subsets{winner}) )
            winner = n;
        end
    end

    % add the winner to the list of covering subsets
    setcover = [setcover winner];

    % remove the winner from the universe
    universe = setdiff(universe, subsets{winner});
end
A.2.2 get_error.m

% Calculate relative and absolute errors of a data set compared with a reference.
% Input:
% - a cell array R containing the reference data (in theory, this would be the "correct" data)
% - a cell array M containing the data with errors that are to be measured
% Note: M must have the same number of cells in the first dimension as R has cells in its only dimension, but M can have any number in the second dimension. Each set of cells in the second dimension will be compared with the set of cells from R's single dimension.
% Output:
% - a matrix of relative errors where each element in the matrix gives the relative error of the corresponding cell in M to the relevant cell in R
% - a matrix of absolute errors where each element in the matrix gives the absolute error of the corresponding cell in M to the relevant cell in R
% Notes: Strict equality will be used when comparing arguments that are not numeric (i.e., if they are equal the result is 0, otherwise it is Inf). If arguments are not numeric then a type mismatch evaluates to Inf. For more details about the equality testing see the isequalwithtolerance function.
% Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)
% This file is part of MATmute.
% MATmute is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
% MATmute is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
% You should have received a copy of the GNU General Public License along with MATmute. If not, see <http://www.gnu.org/licenses/>.

function [rel_errs, abs_errs] = get_error(original, mutated)

num_vars = length(original);
num_mutes = size(mutated, 1);

abs_errs = zeros(num_mutes, num_vars);
rel_errs = zeros(num_mutes, num_vars);

for m = 1:num_vars
    for n = 1:num_mutes
        % Warn if original result (the denominator) is NaN or Inf.
        if any(isnan(original{m}(:)))
            warning('get_error:NaN', 'Reference value of NaN. ')
        elseif any(isinf(original{m}(:)))
            warning('get_error:Inf', 'Reference value of Inf. ')
        end
    end
end
if isequalwithequalnans(original{m}, mutated{n,m})
    % if MATLAB believes they are equal then return 0 error.
    abs_errs(n,m) = 0;
    rel_errs(n,m) = 0;
elseif isequalwithtolerance(original{m}, mutated{n,m}, Inf) && ...
    all(~isnan(mutated{n,m}(:)))
    % isequalwithtolerance is used to check if the two inputs are
    % numeric and of comparable types and sizes. Any comparable
    % numeric arguments will pass the test while any incompatible or
    % non-numeric arguments will fail. Note that non-numeric but
    % equal arguments will have been caught before this point.
    abs_diff = abs(original{m} - mutated{n,m});
    abs_err = max(abs_diff(:));
    rel_err = max(abs_diff(:)/abs(original{m}(:)));
    % If original result (the denominator) is 0 then use absolute
    % error to avoid division by 0.
    if original{m} == 0
        warning('get_error:RelErrDivByZero', 'Absolute error ...
            used instead of relative error to avoid a ... division by 0.')
        rel_err = abs_err;
    end
    % Reserve Inf for obvious faults, for numerical errors that
    % overflow into Inf, use realmax instead.
    if rel_err == Inf
        rel_err = realmax;
    end
    abs_errs(n,m) = abs_err;
    rel_errs(n,m) = rel_err;
else
    % Otherwise return Inf to denote an obvious fault. (This
    % assumes that non-numeric inequalities are obvious.)
    abs_errs(n,m) = Inf;
    rel_errs(n,m) = Inf;
end
end

A.2.3 getkeykillers.m

% Determines which tests are necessary to kill all mutants that exhibit errors
% greater than some kill boundary k_b.
% Inputs:
% - a matrix of error values where each row holds results for a mutant and
%   each column holds results from a test (e.g., data(3,5) indicates the
%   relative error of the 3rd mutant when it is run using the 5th test)
% - the value of the kill boundary k_b
function key_killers = getkeykillers(data, k_b, cutoff, verbose)
    if nargin < 2
        k_b = Inf;
    end
    if nargin < 3
        cutoff = 0;
    end
    if nargin < 4
        verbose = false;
    end
    killed = find ( max (data, [], 2) >= k_b ); % get killed mutants
    if isempty ( killed )
        key_killers = [];
        return
    end
    killers = find ( max (data, [], 1) >= k_b ); % get a list of all tests that kill
    kill_lists = cell (1, length (killers)); % get the a list of mutants that each test kills
    for n = 1:length (killers)
        kill_lists(n) = find ( data(:, killers(n)) >= k_b )'; % get killer n kill
    end
    if verbose
        fprintf ('Calling set cover routine with %i subsets.
', length (killers))
    end
    key_killers = rowcol (setcover (kill_lists)); % key killers are the minimum set of killers that kill (cover) all killed
key_killers = killers(getminsetcover(killed, kill_lists, cutoff, verbose));

% use assertion to confirm that key killers really are sufficient (they
% should be unless there is a bug in the code)
assert(sum(max(data, [], 2) >= k_b) == ...
     sum(max(data(:,key_killers), [], 2) >= k_b))

A.2.4 getkeytests.m

function keys = getkeytests(data, cutoff, verbose)
    if nargin < 2
        cutoff = 1e5;
    end

    if nargin < 3
        verbose = false;
    end

    % get the maximum value for each mutant
    mutant_max = max(data, [1, 2];

    % generate a mask the same size as data that is true wherever a
    % a mutant's max error occurs and false everywhere else
    max_mask = zeros(size(data));
    for n = 1:length(mutant_max)
        max_mask(n,:) = mutant_max(n) > 0 & data(n,:) == mutant_max(n);
    end

% Returns a smaller set of tests that will achieve identical maximum mutant
% error results.
%
% Inputs:
% - a matrix of error values where each row holds results for a mutant and
%   each column holds results from a test (e.g., data(3,5) indicates the
%   relative error of the 3rd mutant when it is run using the 5th test)
% - (optional) a cutoff value to limit the number of combinations that are
%   tested when attempting to find the optimal set of tests; if this cutoff
%   is passed then an approximate solution will be used instead; by default
%   this is 0 which means that the approximate solution is always returned
% - (optional) if true this function will print information about the size
%   of the parameters that are being passed to the set cover function
%
% Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)
% This file is part of MATmute.
% MATmute is free software: you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
% MATmute is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
% You should have received a copy of the GNU General Public License
% along with MATmute. If not, see <http://www.gnu.org/licenses/>.
% use key killer analysis on the max mask with a kill boundary of 1 to % determine the key tests
keys = getkeykillers(max_mask, 1, cutoff, verbose);

% use assertion to confirm that maximum errors for mutants are the same when % only the key tests determined by this function are used; this should % always pass unless there is a coding mistake
assert( all( max(data(:,keys), [], 2) == mutant_max ) )

A.2.5 getminsetcover.m

% Given a set that is to be covered (the "universe") and a collection of subsets % of the universe, this function determines how many of the subsets are needed % to cover the universe.
% Inputs:
% - the "universe", i.e., the set to be covered
% - a 1-dimensional cell array where each cell contains one of the subsets % that are to be used to cover the universe
% - (optional) a cutoff value: if the number of combinations that are to be checked in the current iteration of the function are larger than this % value the the function gives up on finding the optimal solution and uses % an approximate solution that is calculated using the greedy algorithm;
% - a warning is given when the cutoff is exceeded unless this value % is set to 0
% - if set to 0 then the approximation algorithm is used immediately
% - (optional) if true, the function prints the number of combinations that % are being checked in each iteration when that iteration is reached
% Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)
% This file is part of MATmute.
% MATmute is free software: you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
% MATmute is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU General Public License for more details.
% You should have received a copy of the GNU General Public License
% along with MATmute. If not, see <http://www.gnu.org/licenses/>.

function minsetcover = getminsetcover(universe, subsets, cutoff, verbose)

if nargin < 3
    cutoff = 5e6;
end

if nargin < 4
    verbose = false;
end

minsetcover = [];

% list important subsets that may form part of the min set cover
important = 1:length(subsets); % start with all subsets

% while loop is used reduce the number of subsets that must be considered
% by removing obviously extraneous subsets and storing obviously needed
% subsets
old_important = -1;
while (~isequal(important, old_important))
    old_important = important;
    % determine if there are any elements that are covered by all important
    % subsets
    all_covered = universe;
    for n = important
        all_covered = intersect(all_covered, subsets{n});
    end

    % these fully covered elements can be removed from consideration
    universe = setdiff(universe, all_covered); % remove them from universe
    for n = important
        subsets{n} = setdiff(subsets{n}, all_covered); % remove from subsets
    end

    % if universe is now empty then choose any important subset and finish
    if isempty(universe)
        minsetcover = union(minsetcover, important(1));
        return
    end

    % subsets that are subsets of other subsets can be ignored because they
    % are obviously extraneous
    marked = [] ; % track subsets that are to be ignored
    for m = 1:length(important) % loop over important subsets
        if any(marked == m) % if mth subset is already marked
            continue % continue to next m
        end
        for n = m+1:length(important) % loop over important subsets after m
            if any(marked == n) % if nth subset is already marked
                continue % continue to next n
            end
            mUn = union(subsets{important(m)}, subsets{important(n)});
            if isequal(mUn, subsets{important(m)}) % if m U n same as m
                marked = union(marked, n); % n is extraneous so mark
            elseif isequal(mUn, subsets{important(n)}) % if m U n same as n
                marked = union(marked, m); % m is extraneous so mark
                break % m is extraneous so move to next m
            end
        end
    end
    important(marked) = []; % marked subsets are not important so remove

    % find elements that are only covered by one important subset
    counts = zeros(size(universe));
    for n = important
        counts = counts + ismember(universe, subsets{n});
    end
    one_covered = universe(counts == 1);

    % determine which subsets cover single covered elements, add them to the
    % min set cover, and remove them and the elements they cover from
    % further consideration
for n = important
  if ~isempty(intersect(one_covered, subsets{n}))
    minsetcover = union(minsetcover, n);
    important = setdiff(important, n);
    universe = setdiff(universe, subsets{n});
    if isempty(universe) % if nothing left
      return % then done
    end
  end
end

end % end while loop

% make sure that subsets can actually cover the universe
if ~all(ismember(universe, [subsets{[minsetcover important]]}))
  error('Subsets cannot be used to cover universe.')
end

% set subsets to the important subsets; important must be kept as a
% reference to retrieve the actual indices at the end
subsets = {subsets{important}};

% use the greedy algorithm to get an approximate solution, this is used to
% put a lower bound on the size of the optimal solution set
approxsetcover = getapproxsetcover(universe, subsets);

% if cutoff is 0 then return the approximate answer with the subsets that
% have already been identified as part of the min set cover; or if the
% approx set contains only 1 subset then it is optimal so return it as
% solution.
if cutoff == 0 || length(approxtsetcover) == 1
  minsetcover = union(minsetcover, important(approxsetcover));
  return
end

% determine which subset is the biggest; used to calculate lower bound on
% solution size
biggest = 0;
for n = 1:length(subsets)
  if length(subsets{n}) > biggest;
    biggest = length(subsets{n});
  end
end

% the iterations will start with sets that are the size of the lower bound
% which is calculated here and stop at sets that are the same size as the
% approximate solution; we know that the solution must require at least 2
% subsets or it would have already been found
start = max(2, ceil(length(approxsetcover)/harmonic(biggest)));
stop = length(approxsetcover) - 1;

% for faster processing: convert sets and subsets to vectors of logicals
% where nth number in set is represented by "true" in the nth position;
universe = ismember(1:max(universe), universe);
for n = 1:length(subsets)
  subsets{n} = ismember(1:length(universe), subsets{n});
end
subsets = cell2mat(subsets'); % join cells of subsets into one matrix

% columns in subsets where universe vector is false can be deleted because they
% are extraneous; false locations in universe vector can also be deleted
subsets(:,~universe) = [];
universe(~universe) = [];

% subsets will always be needed in the negated form, so negate them now
subsets = ~subsets;

if verbose && start <= stop
    fprintf(' Choosing combinations of %i to %i subsets from %i ...%s
', start, stop, size(subsets,1));
end

% start looking at sets the size of start and try all other combinations up
% to the size of the approximate solution
for k = start:stop

    % get number of possible combinations
    nmb = nchoosek(size(subsets,1), k);

    % if the number of combinations is larger than the cutoff then give up,
    % issue a warning, and use the approximation algorithm
    if nmb > cutoff
        warning('minsetcover:ApproxSolnUsed', 'Problem size exceeded ...
cutoff value so approximate set cover solution used.'
        minsetcover = union(minsetcover, important(approxsetcover));
        return
    end

    if verbose
        fprintf(' Checking %i combinations of %i subsets.
', nmb, k);
    end

    % check if universe can be covered by combinations of k subsets by
    % using recursive subfunction
    cover = checkcmb(universe, subsets, 1, k-1);

    % if a cover was returned then the min set cover has been found;
    % don't forget that already determined subsets must be included
    if ~isempty(cover)
        minsetcover = union(minsetcover, important(cover));
        return
    end
end

% if loop terminates without solution then approximate solution is optimal
minsetcover = union(minsetcover, important(approxsetcover));

end % end main function

% Recursive subfunction that is used to check varying number of combinations
% to see if they cover the universe. Function is designed to execute in a way
% that attempts to reduce the number of repeated calculations.
%
% Inputs:
% - universe to be covered
% - subsets to cover the universe
APPENDIX A.  MATMUTE CODE LISTINGS

\% - index of the first subset to be used as a cover at current depth
\% - maximum recursive depth remaining
\% Output:
\% - list of subsets that cover the given universe; if output is empty then a
cover does not exist
function cover = checkcmb(universe, subsets, start, depth)
    cover = [];
    \% loop through possible combinations of subsets at this depth
    for n = start:size(subsets,1)-depth
        % get universe without the current subset
        newuniverse = universe & subsets(n,:);
        \% if not at end then recursively call this function again
        if depth > 0
            sub_cover = checkcmb(newuniverse, subsets, n+1, depth-1);
        \% if result has a cover then return
            if ~isempty(sub_cover) % if result has a cover then return
                cover = [sub_cover n]; % with current subset included
                return
            end
        \% if at end and the universe is empty then a solution has been found
        elseif ~any(newuniverse)
            cover = n;
            return
        \end
    \end
end % end checkcmb subfunction

\% Calculates the nth harmonic number; used to calculate lower bound on size
\% of the solution.
function S = harmonic(n)
    S = sum(1./(1:1:n));
end

A.2.6  matmute.m

\% Provides an interface to the MATmute mutation testing system.
\% \% Syntax:
\% \% [errors time mutant_outputs original_outputs] = matmute('fn_name',
\% \% \{}TEST1, TEST2, \ldots, TESTN\}, noutarg, 'op1 op2 \ldots', verbose)
\% \%
\% \% Terminology:
\% \% mutation: a change to the code of a program statement
\% \% mutation operator: a function which generates mutations of a statement in a
\% well-defined way
\% \% mutation target: the program (.m file) which is to be mutated
\% \% mutant: a program which differs from the mutation target by one mutation

mutant ID: mutants are sequentially numbered as they are created, this
number is the mutant’s ID

relative error: given a quantity q and its expected value q0, the relative
error er of the quantity is given by

\[ er = \frac{|q - q0|}{|q0|} \]

Description:
This function generates mutations of MATLAB .m files, executes these
mutants using given inputs (i.e., tests), and compares the results with the
results from the execution of the non-mutated file to determine the relative
error in the result of each test.

WARNING: Care should be taken when running the matmute command on .m file
programs which include code that can have permanent effects on the file
system or configuration of a computer. This is because program mutations
can behave in strange and unexpected ways. For example, a program which
normally overwrites a certain file, might overwrite a completely different
file when mutated. The mutator will never introduce new code that
modifies files or configurations so it is safe to use with .m files that do
not currently exhibit any behaviour of that kind.

[errors mute_outs orig_outs time] = matmute('fn_name',
{TEST1, TEST2, ..., TESTN}, noutarg)
generates mutants of function 'fn_name' and uses the input arguments given
in {TEST1, TEST2, ..., TESTN} to run the mutants. Each test is a cell array
containing a valid set of inputs for the target function. The function and
mutants are asked to produce the number of outputs specified by the integer
noutarg.

Because TEST1, TEST2, ..., TESTN are themselves cell arrays, it is important
to understand that the second argument to matmute will be a nested cell
array.

The outputs for mutant M when run using test N can be found in position
(M,N) of the mute_outs cell array and the relative error of each test-mutant
pair can be found in the (m,n) position of the error matrix. The outputs
of the original function on each test are found in the orig_outs cell array.
The time taken relative to the unmutated function is stored in time (note
that this is calculated as the mutant time divided by the original time).

Note that an error score of Inf means that execution aborted or that the
mutant output could not be compared with the target output. A negative
time score means that execution aborted after a relative time equal to the
absolute value of the time score.

'op1 op2 ...' is an optional space delimited list of the mutation operators
which are to be used to create the mutants. If this argument is not given
then the default operators as defined in ops_config.py will be used. For
details about the operators see the Implementation section in this header
or examine the operators.py file. The following operators are currently
implemented and can be selected with their 3-letter ID:

- crp: constant replacement
- neg: branch expression negation
- orp: operator replacement
- sdl: statement deletion

The operators will be applied to each statement in the order which they
are given.
If the verbose flag is set to true then text will be printed listing the current test and mutant that are running.

Remarks:
The computer must be set up to use MATmute. See INSTALL.txt in the MATmute package for instructions.

The mutation target 'fn_name' must be in the current working directory. A folder named 'fn_name'_mutes will be created in this directory during execution. This folder will not be deleted when the mutation testing is completed (although the mutator will overwrite its contents on each run) and must be manually deleted when no longer needed. Therefore, it is suggested that a copy of the mutation target .m file is moved to a designated directory before running MATmute on it. (Don’t forget that MATmute must be called from this directory.)

During the mutation process an unmutated but reformatted and instrumented file is created and named "unmutated.m". The program in this file should be functionally equivalent to the mutation target, but its contents are different for two reasons. Firstly, the mutator requires that the target code be reformatted so that each line of text contains exactly one statement. Comments (except for mutation control comments) are also removed in the reformattting process. Secondly, the code is instrumented with loop monitoring code to count and store the number of times that each loop executes. This is used to kill mutants in which infinite loops may have been introduced.

The results from the unmutated code are compared with the results from the mutation target in order to confirm that the reformattting and instrumentation steps haven’t affected the behaviour of the program. If the outputs from the mutation target depends on timing this test may fail. It may also be possible to write MATLAB code in such a way that the reformattting step misbehaves and forms code which is not equivalent. There are currently no known instances of this kind of misbehaviour, but the parser is not been strongly verified and may make mistakes.

When calling the matmute function it is important to understand that the function’s behaviour may be effected by the number of output arguments that are requested. Some functions behave differently when called with different numbers of storage arguments, and when dealing with these kinds of functions it is important to ensure that the right number of outputs are being requested in order to elicit the desired behaviour of the mutation target.

Unless an attempt is being made to elicit some specific behaviour from the mutation target (as discussed in the previous paragraph), it is usually best to request as many outputs as possible. More outputs will give the mutator more data to examine for noticeable errors.

Implementation:
Mutations are created by applying mutation operators to each statement in the mutation target’s .m file.

The statements are fed, one at a time and in sequential order, to the mutation operators. Each operator returns a list of mutations of the given input statement, and this list is appended to the existing list of mutations which has been formed by the operators that have already run on that statement. The operators execute in the default order defined in
ops_config.py unless the user has specified the mutation operators which
are to be used.

To generate the mutant programs the mutator loops through the statements
of the mutation target. For each mutation (generated by the mutation
operators) of some statement S the mutator generates an .m file where S
has been replaced by that mutation. The mutation in a mutant file (i.e.,
the one change in the code) falls into one of any four types (each mutation
operator is responsible for one type of mutant):

- sdl: a statement is deleted (or commented out)
- neg: a branch condition is negated (forcing the opposite decision)
- crp: hard-coded constants (i.e., digits) in the code are modified
  (e.g., value is incremented by one)
- orp: operators are replaced by another operator from the same class
  (e.g., '+' -> '*', '&', -> '|' or '<' -> '==')

For a full understanding of the mutation operators effects see the
operators.py file which is part of the MATmute package. New operators can
be added by modifying operators.py and the behaviour of existing operators
can be modified by modifying operators.py or ops_config.py. The parameters
in the ops_config.py affect the behaviour of some of the operators and are
intended to allow easy customization of the behaviour of some operators.

Further details about the implementation of the mutator can be determined
by examining the code in the .py files in the MATmute package.

Copyright 2008 Daniel Hook (dan_hook@users.sourceforge.net)

This file is part of MATmute.

MATmute is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

MATmute is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with MATmute. If not, see <http://www.gnu.org/licenses/>.

function [errors time mute_outputs original_results] = matmute(fn_name, ...
inarg, noutarg, op_list, verbose)

% Given timing factor F and unmutated loop iterations I, if a mutant’s
% loops use more than F*I iterations then the action specified by the
% fail_on_bad_loop flag (see below) will be taken.
loop_timing_factor = 100;

% If fail_on_bad_loop is set to true then loops which exceed the tick
% limit will be fail with an error. Otherwise, the loop will be exited
% but the program will execute until completion.
fail_on_bad_loop = true;

% Required declarations for running the mutations.
global MUTE_ticks MUTE_ticklimit MUTE_testcnt % used for loop monitoring
MUTE_ticklimit = []; % empty list means timing limits not yet determined
% Create output storage string. This is used whenever calling a function
% with the number of storage arguments used given in noutarg.
outarg_str = '[';
for k = 1:noutarg
    outarg_str = [outarg_str ' out{' num2str(k) '} '] ;
end
outarg_str = [outarg_str '] = '] ;

% Check for input errors.
if nargin < 3
    error('At least three input arguments are required.')
elseif ~ ischar(fn_name)
    error('First input argument must be a string.')
elseif ~ iscell(inarg)
    error('Second input argument must be a cell array.')
elseif ~ isnumeric(noutarg)
    error('Third input argument should be a whole number.')
end

% If op_list not given then set it as empty (for default behaviour).
if nargin < 4
    op_list = [] ;
end

% If verbose flag not given then set to false.
if nargin < 5
    verbose = false;
end

% If function's mutant directory exists then erase it's contents so that
% the new mutants will not be mixed with old mutants.
if isdir([pwd '/' fn_name '_mutes'])
    rmdir([fn_name '_mutes'], 's')
end

% All functions in memory are cleared before starting, this avoids
% problems related to using old versions of files that are still stored
% in memory.
clear functions

initdir = pwd ; % store the current directory
initpath = path ; % store current path so it can be restored
initwarning = warning('query', 'all');

% Generate the call string which calls the mutator.
callstring = ['python -m matmute ' fn_name];

% If operators selected then append the active operator list.
if ~ isempty(op_list)
    callstring = [callstring ' -o '' op_list ''];
end

% If fail_on_bad_loop is true then append "--inferr" flag to call string.
if fail_on_bad_loop
    callstring = [callstring ' --inferr'];
end

% Call the mutator using the callstring.
[status output] = system(callstring);

% Check if the mutator reported any errors.
if status == 0
    disp('Creation of mutants complete."
else
    disp(output);
    error('Mutator encountered a problem when creating the mutants. ...
         See error message above.')
end

% Parse the mutant IDs from the mutator’s output string.
mute_ids = regexp(deblank(output), '\n', 'split');

% If empty string is returned as the mutant IDs then quit.
if isempty(mute_ids)
    error('No mutants created. Perhaps you should try different ...
          mutation operators?"
else
    fprintf('Tests will be executed on %i mutants.\n', length(mute_ids))
end

% Pre-allocate variables for mutation results and analysis.
original_results = cell(length(inarg), 1);
unmute_time = zeros(length(inarg), 1);
errors = zeros(length(mute_ids), length(inarg));
time = zeros(length(mute_ids), length(inarg));
mute_outputs = cell(length(mute_ids), length(inarg));

% Execute tests on the original version of the function and store results.
for n = 1:length(inarg)
    if verbose
        fprintf('Running test %i on original function.\n', n)
    end
    try
        eval([outarg_str fn_name '{inarg{n}:}';']);
        original_results(n) = out;
    catch
        error('Test %i did not run on the original function.', n)
    end
end

% Add the current directory to the path (so that any functions in this
% directory are still accessible).
path(initdir, path);

% Change to the directory where the mutants are stored.
cd(['fn_name '_mutes']);

% Use try-catch to ensure that MATLAB returns to the original directory.
try
    % MUTE_testcnt counts the number of tests that have been executed on
    % each mutant. It is used by the mutated files to determine which
    % ticklimit corresponds to each test, and must be reset every time a new
    % mutant is being analysed.
    MUTE_testcnt = 0;
end
% Execute tests on the unmutated function and compare the results with
% the results from original function.
for n = 1:length(inarg)
  if verbose
    fprintf('Running test %i on instrumented function.\n', n)
  end
  tic; % reset timer
  eval([outarg_str 'unmutated(inarg{n}{:}); ']);
  unmute_time(n) = toc;
  if ~isequalwithequalnans(original_results{n}, out)
    error('Mutator fault: instrumented (but unmutated) version ...
    gives different results than the original version.')
  end
end

% Use results from the instrumented version for loop iteration limits.
% (+1 ensures that ticklimit is nonzero.)
MUTE_ticklimit = (MUTE_ticks+1)*loop_timing_factor;

% Loop through all the mutants.
for m = 1:length(mute_ids)
  MUTE_testcnt = 0; % as above, MUTE_testcnt must be reset
  % Turn off warnings (except desired mutant warnings) while
  % running mutants.
  warning('off', 'all')
  warning('on', 'mutant:TickLimitExpired')
  warning('on', 'get_error:RelErrDivByZero')
  warning('on', 'get_error:NaN')
  warning('on', 'get_error:Inf')
  % Begin execution of the mutants using the tests in inarg.
  for n = 1:length(inarg)
    if verbose
      fprintf('Running test %i on mutant %i.\n', n, m)
    end
    try
      tic; % reset timer
      eval([outarg_str mute_ids{m} '(inarg{n}{:}); ']);
      mute_time = toc; % get execution time
      % If error caught then store message and location.
      catch
        e = lasterror; s = e.stack(1);
        for k = 1:noutarg
          out{k} = sprintf('_MUTE_err caught originating from ...
          line %d of "%s":\n%s', s.line, s.name, e.message);
        end
        mute_time = -toc; % (-) indicates error time
      end
      mute_outputs{m,n} = out;
      errors(m,n) = max(get_error(original_results{n}, out));
      time(m,n) = mute_time/unmute_time(n);
    end
    % Turn warnings back on.
    warning(initwarning)
  end
end
end

catch
    % Return to main directory, reset variables and pass error along.
    cd(initdir);
    path(initpath);
    warning(initwarning)
    clear global MUTE_ticks MUTE_ticklimit MUTE_testcnt
    rethrow(lasterror)
end

% Change back to the original directory and remove addition to the path.
% cd(initdir);
% path(initpath);

% Reset global variables.
clear global MUTE_ticks MUTE_ticklimit MUTE_testcnt

% Display success message.
disp('Tests finished executing.')

end
Appendix B

Mutation Target Code Listings

The following eight functions were used as mutation targets in the experiments discussed in this thesis.

The codes, except for \texttt{nwtsqrt}, are based on MATLAB functions that are given and/or described in [29].

B.1 \texttt{binSearch.m}

```matlab
function ia = binSearch(x, xhat)

n = length(x);
if xhat < x(1) | xhat > x(n)
    error(sprintf('Test value of %g is not in range of x', xhat));
end

ia = 1;
ib = n; % Initialize lower and upper limits
while ib-ia>1
    im = fix((ia+ib)/2); % Integer value of midpoint
    if x(im) < xhat
        ia = im; % Replace lower bracket
    else
        ib = im; % Replace upper bracket
    end
end % When while test is true, ia is desired index
```

B.2 \texttt{gaussQuad.m}

```matlab
function I = gaussQuad(fun,a,b,npanel,nnode,varargin)

[z,wt] = GLNodeWt(nnode); % compute the nodes and weights
H = (b-a)/npanel;
H2 = H/2;
x = a:H:b;
I = 0;
```
for i = 1:npanel
    xstar = 0.5*(x(i)+x(i+1)) + H2*z; % Evaluate 'fun' at these points
    f = feval(fun,xstar, varargin{:}); % Add contribution of this subinterval
    I = I + sum(wt.*f); % Add contribution of this subinterval
end
I = I*H2; % Factor of H/2 for each subinterval

function [x,w] = GLNodeWt(n)
    beta = (1:n-1)./sqrt(4*(1:n-1).^2 - 1);
    J = diag(beta,-1) + diag(beta,1); % eig(J) needs J in full storage
    [V,D] = eig(J);
    [x,ix] = sort(diag(D)); % nodes are eigenvalues, which are on diagonal of D
    w = 2*V(1,ix)'.'^2; % V(1,ix)' is column vector of first row of sorted V

B.3 GEPiv.m

function x = GEPiv(A,b)
    ptol = 50*eps;
    [m,n] = size(A);
    nb = n+1;
    Ab = [A b]; % Augmented system
    % --- Elimination
    for i = 1:n-1 % loop over pivot row
        [pivot,p] = max(abs(Ab(i:n,i))); % value and index of largest available pivot
        ip = p + i - 1; % p is index in subvector i:n
        if ip ~= i % ip is true row index of desired pivot
            Ab([i ip],[ :) = Ab([ip i],:); % perform the swap
        end
        pivot = Ab(i,i);
        if abs(pivot)<ptol,
            error('zero pivot encountered after row exchange');
        end
        for k = i+1:n % k = index of next row to be eliminated
            Ab(k,i:nb) = Ab(k,i:nb) - (Ab(k,i)/pivot)*Ab(i,i:nb);
        end
    end
    % --- Back substitution
    x = zeros(n,1); % preallocate memory for and initialize x
    x(n) = Ab(n,nb)/Ab(n,n);
    for i=n-1:-1:1
        x(i) = (Ab(i,nb) - Ab(i,i+1:n)*x(i+1:n))/Ab(i,i);
    end

B.4 nwtsqrt.m

function y = nwtsqrt(x, init)
    tol = 1e-10;
    y = init;
    while abs(y*y - x) > tol
\[ y = (y + x/y)/2; \]

\section*{B.5 odeRK4.m}

\begin{verbatim}
function [y] = odeRK4 (diffeq,tn,h,y0)
t = (0:h:tn)'; % Column vector of elements with spacing h
n = length(t); % Number of elements in the t vector
y = y0*ones(n,1); % Preallocate y for speed
h2 = h/2;
h3 = h/3;
h6 = h/6; % Avoid repeated evaluation of constants

% Begin RK4 integration; j=1 for initial condition
for j =2: n
  k1 = feval(diffeq, t(j -1), y(j -1) );
  k2 = feval(diffeq, t(j -1)+h2, y(j -1)+h2*k1 );
  k3 = feval(diffeq, t(j -1)+h2, y(j -1)+h2*k2 );
  k4 = feval(diffeq, t(j -1)+h, y(j -1)+h*k3 );
  y(j) = y(j -1) + h6*(k1+k4) + h3*(k2+k3);
end
\end{verbatim}

\section*{B.6 powerit.m}

\begin{verbatim}
function [mu] = powerit (A, nit)
[m,n] = size(A);
x0 = ones(m,1);

u = x0;
for k =1:nit
  u = A*u;
  mu = norm(u, inf);
  u = u/mu;
end
\end{verbatim}

\section*{B.7 simpson.m}

\begin{verbatim}
function I = simpson (fun,a,b,npanel)
n = 2*npanel + 1; % total number of nodes
h = (b-a)/(n-1); % stepsize
x = a:h:b; % divide the interval
f = feval(fun,x); % evaluate integrand
I = (h/3)*( f(1) + 4*sum(f(2:2:n-1)) + 2*sum(f(3:2:n-2)) + f(n) );
\end{verbatim}

\section*{B.8 sphereFnet.m}

\begin{verbatim}
function out = sphereFnet(v, m, d, g, p, u)
\end{verbatim}
Re = p*v*d/u;

cd = calcCd(Re);

out = m*g - cd*0.5*p*v^2*pi*d^2/4;

function cd = calcCd(Re)

data = load('sphereCd.dat');

if Re < 0
    error('Negative Re value encountered')
elseif Re <= 2e4
    cd = 24/Re + 6/(1+sqrt(Re)) + 0.4;
elseif Re <= 3.99e6
    cd = interp1(data(:,1), data(:,2), Re);
else
    cd = 0.1810;
end

% sphereCd.dat contains the following:
% 20000  0.4430
% 38200  0.4900
% 73000  0.5030
% 144000  0.5150
% 220000  0.5080
% 258000  0.4940
% 293000  0.4740
% 319000  0.4360
% 341000  0.3740
% 350000  0.2580
% 365000  0.1260
% 384000  0.0836
% 417000  0.0692
% 462000  0.0655
% 548000  0.0726
% 743000  0.0888
% 1190000  0.1230
% 2050000  0.1570
% 3220000  0.1740
% 3990000  0.1810