Test-Driven Development with `seamless`

Paul Adamson

January 1, 2015

Contents

1 Introduction **5**
1.1 Test-Driven Development 5
1.1.1 A Typical TDD Process 5
1.1.2 TDD Aids Design 6
1.1.3 Tests as Code Documentation 6
1.2 Literate Programming 7
1.2.1 The `seamless` Framework 8
1.3 `seamless` Package 9

2 Notation **11**

3 Organization **13**

4 Requirements **15**
4.1 Scope 15
4.2 Functional Requirements 15

5 Rectangle Integration **17**
5.1 Left Rectangle Method 17
5.2 Right Rectangle Method 24
5.3 Midpoint Rectangle Method 25

6 Trapezoid Integration **27**

7 Simpson's Rule Integration **29**

Index **35**

Bibliography **37**

1 Introduction

Within the `seamless` framework for science code development, a test-driven development approach is combined with concepts from literate programming[3] and best practices in software engineering. The resulting framework encourages development of very modular, well-documented code that is robust and reusable. Currently, the framework supports the Chapel (`http://chapel.cray.com`) programming language, but extension to other languages is straightforward. This extensibility is possible because the framework is designed around the LaTeX typesetting system. When following the `seamless` approach, the developer is essentially writing a LaTeX document containing the requirements, documentation, specification, test suite, and source code. Tests and source code are extracted and the tests are run by executing a script that is provided with the framework.

In addition to this introductory chapter, the remainder of the tutorial is an example `seamless` specification for a simple software package to compute the integral of a one-dimensional function using various numerical integration techniques. More specifically, the tutorial solves the Rosetta Code numerical integration task[1] in Chapel. As you will see, the "off the shelf" format of a `seamless` specification supports colored text boxes to capture various aspects of the specification such as *Notes*, *Rationales*, and *Futures*. In addition, the tutorial contains an additional type of text box not normally present in a specification to explain background information about the `seamless` framework in context of the tutorial. Here is an example of one of these special text boxes used only in the tutorial:

> *`seamless`*. Example text box used to provide background on the `seamless` approach in context of the tutorial.

But before we get to the example use of `seamless`, some background on test-driven development, literate programming, and software engineering practices is in order.

1.1 Test-Driven Development

Test-driven development, or TDD, is the notion that developers will improve both the design and accuracy of their code by *writing the test* for a particular feature *before writing the code* that implements the feature according to the specification. In other words, the TDD process begins with writing an automated test for code that does not yet exist. After a test is written for a particular feature defined in the specification, the programmer then writes the implementing code to get the test to pass. This process is repeated until all features in the specification are implemented. The idea is that by writing tests before code, rather than after, the tests will help guide the design in small, incremental steps. Over time, this creates a well-factored and robust codebase that is easier to modify.

1.1.1 A Typical TDD Process

A typical TDD process goes something like this:

1. Create a test. The test should be short and test for only one code feature. The test should run automatically.

2. Make sure the test fails. Verifying test failure before writing code helps to ensure that you are indeed testing the intended feature.

3. Write the simplest code possible to make the test pass. The code doesn't necessarily need to be 'good' yet. Try not to look ahead too far. Write just enough code to clear the current error.

4. Refactor the code. Once the test passes, improve the code through refactoring. Clean up duplication. Optimize. Perhaps implement some parallelization. Consider creating new abstractions or objects. Refactoring is a key part of design, so spend an appropriate amount of time on this step.

5. Retest. Run the tests again to make sure you haven't changed any behavior.

6. Retest and refactor some more. Do you get the point on refactoring yet?

Repeat the above cycle until your code is complete. This will, in theory, ensure that your code is always as simple as possible and completely covered by tests.

1.1.2 TDD Aids Design

Test-driven development aids software design in several ways. Deciding which test to write next forces you to think about what functionality you need in your code in a systematic way. Deciding how to test your code for a given functionality forces you to think about how that functionality should be implemented, driving you to consider how the rest of the code will interface with it. After the test passes, you refactor and refactor again, improving the design of the code in incremental ways. Since you have working unit tests for all of the functionality that has been implemented prior to a given refactoring stage, you can feel confident that the improvements you make through refactoring will not break another part of the code.

Continual alignment of source code with relevant tests as described above tends to ensure that the resulting software package is made up of small functions and objects that are loosely coupled and have minimal side effects. TDD encourages a good code structure with low coupling (different parts of the code have minimal dependencies on each other) and high cohesion (code that is in the same unit is all related). Code written without tests will tend to have low cohesion and high coupling characteristics, making it much harder to cover with tests written after development.

1.1.3 Tests as Code Documentation

A case can be made in some domains (*e.g.* web development) that automated test suites provide an alternate means of documenting code—that the tests are, in essence, a detailed specification of the code's behavior. This is somewhat true in technical computing, but full documentation of scientific and engineering software requires more than just brief comments and example output. Surely, documentation for a function that computes the electron-electron repulsion integral in a quantum chemistry code must have some description of the type of electronic wavefunction for which the code is valid!

1.2 Literate Programming

Enter stage right...literate programming.

A typical computer program consists of a text file containing program code. Strewn throughout will likely be scant plain text descriptions separated out by “comment delimeters” that document various aspects of the code. Since the actual code itself is presented in a such a way that supports the syntax, ordering, and structure that the programming language (and hence compiler) requires, the code comments will be relatively disorganized and disjointed if you are reading them for documentation purposes. The way a code suite is organized in source is generally much different than the way thorough documentation is developed. The plain text nature of the comments also greatly limits their information value.

In literate programming the emphasis is reversed. Instead of writing *a lot of* code that contains *some* plain text documentation, the literate programmer writes *thorough, well-organized, and content-rich* documentation that contains *modular and efficient* code. The result is that the commentary is no longer hidden within a program surrounded by comment delimiters; instead, it is made the main focus. The “program” becomes primarily a document directed at humans, with the code interspersed within the documentation, separted out by “code delimiters” so that it can be extracted out and processed into source code by literate programming tools. The nature of literate programming is summarized pretty well in a quote from the online documentation for the FunnelWeb literate programming preprocessor:

> “The effect of this simple shift of emphasis can be so profound as to change one’s whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick.”
>
> -FunnelWeb Tutorial Manual[5]

The following list of requirements can be used to define a “literate program:”[2]

1. The high-level language code and the associated documentation come from the same set of source files.
2. The documentation and high-level language code for a given aspect of the program should be adjacent to each other when presented to the reader.
3. The literate program should be subdivided in a logical way.
4. The program should be presented in an order that is logical from the standpoint of documentation rather than to conform to syntactic constraints of the underlying programming language(s).
5. The documentation should include notes on open issues and future areas for development.
6. Most importantly, the documentation should include a description of the problem and its solution. This should include all aids such as mathematics and graphics that enhance communication of the problem statement and the understanding of its challenge.
7. Cross references, indices, and different fonts for text, high-level language keywords, variable names, and literals should be reasonably automatic and obvious in the source and the documentation.
8. The program is written in small chunks that include the documentation, definitions, and code.

The documentation portion may be any text that aids the understanding of the problem solved by the code (*e.g.* description of the algorithm that is implemented). The documentation is often significantly longer than the code itself. Ideally, the problem is described in a way that is agnostic of the language in which the code is written. For example, documentation for code that integrates a function $f(x)$ would have discussion of discontinuities, various integration methods available (*e.g.* trapezoidal, Simpson), domain of integration, etc. In addition to basic shortfalls in documentation and testing in scientific codes, a recent study highlighted the widespread lack of basic context in available documentation.[4] Literate programming solves this problem, ensuring that context is created while the program is written.

Test-driven development and literate programming are certainly compatible. In fact, they are complementary and their combined use is a rare actual example of "the whole is greater than the sum of its parts," especially in the context of developing science code. In one document, we can clearly outline the problem to be solved, develop a test for the code that we want, and document the code that solves the problem. As this is done in an incremental manner, the scientist develops the code that solves the right problem in an efficient and robust manner. As will be seen below, the proccess also supports several fundamental aspects of good software engineering.

1.2.1 The `seamless` Framework

The `seamless` framework begins first with writing a high-level requirements specification. Failing to write down the requirements is the single biggest unnecessary risk a developer can take in a software project, resulting in greatly diminished productivity. For any non-trivial project (more than a few days of coding for one programmer), the lack of a thorough requirements specification will always result in more time spent and lower quality code. Even for trivial examples, a short, informal specification will at least help to ensure accuracy of the resulting code.

Depending on the size and complexity of the code, the requirements specification can have multiple levels (or *layers*). It clearly defines the problem that the program will solve. At some point in development of the requirements specification, the developer should evaluate available algorithms and consider how data produced from the program will be used. Even if a specification is written solely for the benefit of a lone developer, the act of writing the specification—describing how the program works in detail—will force the developer to begin thinking about design of the program.

Once a high-level requirements specification is in hand, the remainder of the `seamless` process is used to decompose the requirements, document the solution, write tests, and write source code:

1. Document a part of the problem and its solution.
   (a) Describe a small part of the problem to be solved. The description should include all aids such as mathematics and graphics that enhance communication of the problem statement and the understanding of its challenge.
   (b) Solve the problem, again using all aids at your disposal (*e.g.* math, graphics).
   (c) Include appropriate references to higher level requirement specifications.
2. Create a test.
   (a) The test should be as short as possible and test for one solution in your overall problem.[1]

[1]Note here that "one thing in your code" is replaced with "one solution in your overall problem." This change emphasizes the literate programming emphasis on documenting the problem and solution before writing code. Writing the test is another form of documenting the solution.

   (b) The test should run automatically.
   (c) Make sure the test fails.

3. Create the code.
   (a) Write the simplest code possible to pass the test.
   (b) After the test passes, refactor to improve the code.
   (c) Run the tests again to make sure the code still passes.
   (d) Refactor and retest some more.

Repeat the above cycle until your code is complete. In theory, the resulting code will have the following characteristics:

- completely documented
- simple
- readable
- completely covered by tests
- robust
- accurate
- maintainable
- reusable

1.3 `seamless` Package

The following files make up this tutorial for `seamless`:

Makefile supports ‘pdflatex‘ and ‘latex‘ to compile the LaTeX package into the tutorial pdf; also has targets to make source code and tests from the LaTeX package and run the tests (usage of these targets is described in the tutorial)

seamless.cls provides the ‘seamless‘ LaTeX document class which is a modification of the ‘book‘ class

seamless.sty provides several environments and commands used in the seamless approach (usage is described in the tutorial)

Numerical_Integration.tex the main LaTeX document with includes for the remaining LaTeX files (see Chapter 3, Organization, for details on the structure of the tutorial)

chapel_listing.tex used with the LaTeX ‘listings‘ package to properly format Chapel code

references.bib contains bibliography entries for use by ‘bibtex‘

Adapting `spec.tex` and the associated LaTeX files for a new software project is straightforward. Once you've adapted the structure of the LaTeX package in the `\spec` directory for your purposes, and you've written a decent requirement specification, you're ready to begin the process described in Section 1.2.1.

As I stated above, the `seamless` approach is "quasi-literate" programming. While the approach that I've described meets the intent of the requirements outlined in Section 1.2 above, it fails to fully implement one of the two main concepts of literate programming.[3] The first concept, described at length above, is that code should have good documentation with all of the supporting mathematics and graphics necessary to convey its function.

The other main concept of literate programming is that the best order to explain the parts of a program is not necessarily going to be the same order that the compiler needs to process the code. For example, you might have

```
proc readInAtoms(filename:string) {
  var infile = open(filename, iomode.r);
  var reader = infile.reader();

  // 55 lines of error handling code

  readNuclei(reader);
  readBasis(reader);

}
```

When first describing the function of the above block of code, the developer wants to focus on a description of opening the file and reading in data, not discussing the error handling just because the computer language requires it to be in between the open and the read. You probably prefer to discuss the main logic first, returning to the error-handling part at some later point in the documentation, perhaps in a section of the documentation that covers error-handling for the entire software package.

Also, for a collaborator that is reviewing code to understand and perhaps contribute to it, having all of that error handling present in the first encounter with the code block is very distracting. It is an impediment to understanding the main purpose of the code.

TODO. Reword next paragraph and describe how the seamless approach deals with it (presenting evolutions of the code and only using the latest one).

Knuth's idea goes right to the heart of the problem. When you program in a literate programming system, you get to write the code in any order you want to. The literate programming system comes with a utility program, usually called `tangle`, which permutes the code into the right order so that you can compile or execute it. Perl doesn't have anything like tangle. You can write comments and typeset them with your favorite typesetting system, but you still have to explain the code in an order that makes sense for the perl interpreter, and not for the person who's trying to understand it.

2 Notation

Three types of code chunks that make up the software package are presented in this specification and delineated with the appropriate keyword in italics: *Source, Helper, or Test*. The filename containing the code chunk is given in parenthesis following the keyword. A brief description of the code chunk is also listed. The actual code is represented with a fixed-width font where keywords are bold and comments are italicized. An example helper code chunk is listed below.

Helper (testFunctions.chpl). Provide the functions used in the tests.

```
proc f1(x:real):real {
  return x**3;
}
proc f2(x:real):real {
  return 1/x;
}
proc f3(x:real):real {
  return x;
}
```

Examples of how to use the software are also provided and delineated with the keyword *Example* followed by a description of the use case. Here is an example of an *Example*:

Example. The following line of code calls the function `leftRectangleIntegration` to perform the integral of the function `f1` over the interval $[1.0, 4.0]$ using the left rectangle method with 100 subdivisions and stores the result in the variable `result`.

```
var result: real = leftRectangleIntegration(a = 1.0, b = 4.0, N = 100, f = f1);
```

Different color text boxes are used to highlight various types of items throughout the specification.

TODO. Things that need to be done for this version of the software.

Note. Something of note that does not fit into any other category.

Rationale. An explanation for a particular design choice.

Open issue. Issue that we do not know how to handle.

Future. Issue or feature that we have a story about, but which is not yet fully-designed or implemented.

3 Organization

This tutorial is organized as follows:

Chapter 1 Tutorial Introduction, background on the test-driven development process of the `seamless` package, the importance of starting with good requirements, and a process to establish traceability of source code to requirements.

Chapter 2 Notation, Introduces the notation that is used throughout a typical `seamless` specification.

Chapter 3 Organization, describes the contents of each of the chapters within this tutorial.

Chapter 4 Requirements, scope and functional requirements for the numerical integration code that we will develop in the tutorial.

Chapter 5 Rectangle Integration, documentation, source code, and test suite for implementation of the rectangle method of numerical integration in the Chapel language.

Chapter 6 Trapezoid Integration, documentation, source code, and test suite for implementation of the trapezoid method of numerical integration in the Chapel language.

Chapter 7 Simpson's Integration, documentation, source code, and test suite for implementation of the Simpson's method of numerical integration in the Chapel language.

4 Requirements

seamless. As described in Section 1.2.1, we must begin with good requirements. In the example shown below, we begin with a scope that is a brief description of the software package, summarizing the code's high-level capabilities. The functional requirements then spell out the specific requirements in sufficient detail that every line of code can be traced back to a labeled item (*e.g.* **R1.1**). The convention used below is that similar requirements are nested together, and only the most deeply nested items are numbered in a given chain of parent/child nestings. Each labeled item inherits the language of all of its higher level parents. For example, in the requirements list

The code shall take inputs `a` and `b`

and `c`

R1.1 and compute `a + b - c`

R1.2 and compute `a - b + c`

R2 and compute `a * b`

the Requirement **R1.1** is "the code shall take inputs `a` and `b` and `c` and compute `a + b - c`; however, the Requirement **R2** is "the code shall take inputs `a` and `b` and compute `a * b`.

To place a requirement label, use the command `\req{x}`, where `x` is the desired number (*e.g.* `\req{1.1}`).

4.1 Scope

The scope of this application is the numerical integration of arbitrary functions to solve the Rosetta Code numerical integration task[1] in Chapel. Solving the task requires development of functions to calculate the definite integral of a function ($f(x)$) using rectangular (left, right, and midpoint), trapezium, and Simpson's methods.

4.2 Functional Requirements

The code shall have functions to calculate the definite integral of a function ($f(x)$).

Available methods of integration shall include:

R1.1 left rectangular

R1.2 right rectangular

R1.3 midpoint rectangular

- **R1.4** trapezoid
- **R1.5** Simpson's

R2 The integration functions shall take in the upper and lower bounds (a and b) and the number of approximations to make in that range (N).

R3 The integration functions shall return the value for the integral.

The test suite shall demonstrate the code's capability by showing the results for the following cases:

- **R4.1** $f(x) = x^3$, where x is $[0, 1]$, with 100 approximations. The exact result is 1/4, or 0.25.
- **R4.2** $f(x) = 1/x$, where x is $[1, 100]$, with 1,000 approximations. The exact result is the natural log of 100, or about 4.605170.
- **R4.3** $f(x) = x$, where x is $[0, 5000]$, with 5,000,000 approximations. The exact result is 12,500,000.
- **R4.4** $f(x) = x$, where x is $[0, 6000]$, with 6,000,000 approximations. The exact result is 18,000,000.

5 Rectangle Integration

`seamless`. Suggestions for Writing Your Specification

- There should be example programs for every package concept and for every subsection.

- In explaining a feature, try to address everyone in your audience (e.g. the package developers as well as the black box users). Explain the syntax of the feature and any implementation details. Also explain the impact of the feature and how it is used when piecing together an application.

- Italicize the term when you are defining it. Be sure to add it to the index. You want as many things as possible in the index. (See below for guidance on index term creation.)

Index Term Creation ——————-

- It is better to err on the side of too many entries; you can cull excessive entries later.

- Double-billing is OK, (e.g. ‘‘ and ‘‘)

- An acceptable exception is when these are visually close, so would be noisy (e.g. ‘‘ and ‘‘) In this case, the second form is preferred.

- Global topics can appear at the top level; otherwise, they are qualified by the chapter name (or general topic).

The rectangle method computes an approximation to a definite integral by finding the area of a collection of rectangles whose heights are determined by the values of the function. Specifically, the interval $[a, b]$ over which the function is to be integrated is divided into N equal subintervals of length $h = (b - a)/N$. The rectangles are drawn with one base along the x-axis. Depending on whether the method is left, right, or midpoint, the left corner, right corner, or midpoint, respectively, of the side opposite the base lies on the graph of the function. The approximation to the integral is then calculated by adding up the areas (base multiplied by height) of the N rectangles, giving the formula:

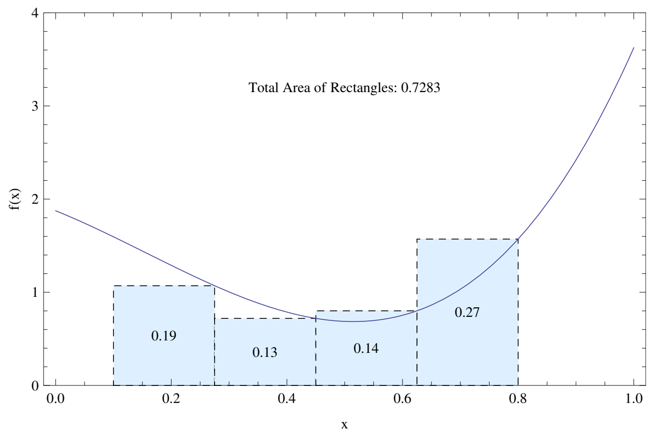
$$\int_a^b f(x)dx \approx h \sum_{n=0}^{N-1} f(x_n) \tag{5.1}$$

where

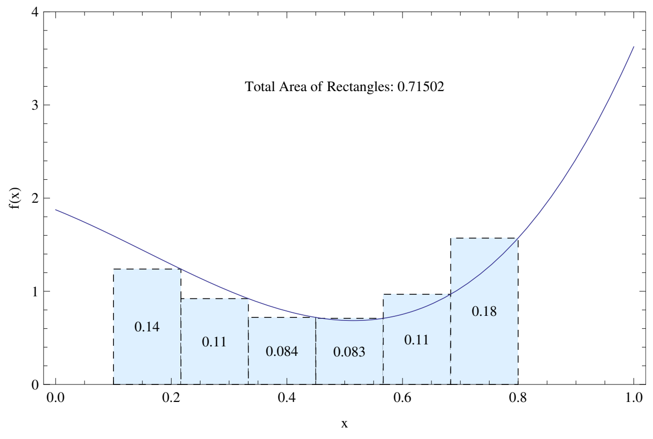
$$h = (b - a)/N \tag{5.2}$$

The formula for x_n for the left, right, and midpoint methods are given in Table 5.1. As N gets larger, the rectangle method becomes more accurate. This is illustrated in the series of plots in Figure 5.1.

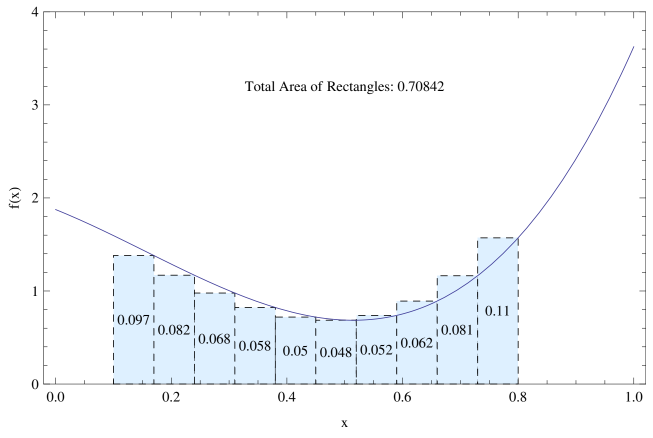
5.1 Left Rectangle Method



(a) $N = 4$



(b) $N = 6$



(c) $N = 10$

Figure 5.1: Numerical integration of $f(x) = (2x-0.5)^3 + (1.5x-1)^2 - x + 1$ for x in $[0.1, 0.8]$ by the (right) rectangle method for increasing values of N. The number inside each rectangle is the area of that rectangle, and the total area is displayed on each graph. The exact value of the integral is 0.70525.

Table 5.1: Formula for x_n in Equation 5.1 of rectangle numerical integration methods.

Method	x_n
left	$a + nh$
right	$a + (n+1)h$
midpoint	$a + \left(n + \frac{1}{2}\right) h$

`seamless`. Now that we've described the rectangle methods fairly well, we will begin developing the code for the left rectangle version. The first step is to develop the tests, and every test needs an expected value against which to compare. For our cases, we have the exact values for the integrals, but exact values alone are insufficient to test numerical methods against. We need to understand by how much the output from our functions can deviate from the exact values. Fortunately, the maximum error expressions for these methods are well known.

If $f(x)$ is increasing or decreasing on the interval $[a, b]$, the maximum error E for left or right rectangular numerical integration is given by

$$E \leq \frac{b-a}{N} \left| f(b) - f(a) \right| \tag{5.3}$$

We can create a helper function to compute the maximum error for left and right rectangle methods using Equation 5.3. The calculated value will be used in tests for the left and right rectangle methods to check that the result is within the maximum error expected for a given a, b, and N.

S1 Helper function `leftRightRectangleMaxErr` returns the maximum error expected for left or rectangle method numerical integration. It takes in a reference to a pre-defined function f, the bounds a (real) and b (real) of the interval for definite integration, and the number N (integer) of subintervals used. The function will be entered in `leftRightRectangleMaxErr.chpl`.

Helper (leftRightRectangleMaxErr.chpl).

```
proc leftRightRectangleMaxErr(a: real, b: real, N: int, f): real{
  return ((b-a)/N)*abs(f(b)-f(a));
}
```

`seamless`. In `seamless` vernacular, the helper files are chunks of code that are used to support testing that the developer wants to have outside of the tests. The most likely reason being that the code contains setup or auxiliary functions that are used for multiple tests. In our example above, we are using some foresight and envisioning that the `leftRightRectangleMaxErr` function will also be used in a test for the left rectangle numerical integration function. To extract the helper files from your latex source files, run the following command in the same directory as your latex source:

```
[./tutorial/] $ make helpers
```

This command runs the `helpers` target in the Makefile at the root of the tutorial directory (`./tutorial/Makefile`. A Makefile is a text file written in a certain prescribed syntax. Together with the `make` utility, it helps automate repetitive commandline tasks such as building software from its source files. In this case, the `helpers` target cleans out the `./tutorial/helper` directory and executes the `./util/extract\_helpers` python script with the appropriate arguments.

One of the functions that we need to test our methods against is $f(x) = x^3$, with $a = 0$, $b = 1$, and $N = 100$. Since the function is increasing on the interval $[0, 1]$, we can use the helper function that we just created to compute the maximum expected error. We are ready to create our first test for a function that we will write to compute the definite integral using the left rectangle method. This function will be called `leftRectangleIntegration` and will be written to `leftRectangleIntegration.chpl`. Since we know we have four tests to construct (Requirements R4.1 through R4.4), we will label the specification for this first test **S2**

TODO. Add seamless note on how to reference spec's and req's.

S2 Test `leftRectangleIntegrationTest1.chpl` loads modules `leftRightRectangleMaxErr` and `leftRectangleIntegration`. It defines a function `f` that takes x (real) and returns x^3 (real). It passes $a = 0.0$, $b = 1.0$, $N = 100$, and `f` to the function `leftRightRectangleMaxErr` and stores the result in the variable `maximumError` (real). It passes $a = 0.0$, $b = 1.0$, $N = 100$, and `f` to the function `leftRectangleIntegration` and stores the result in the variable `calculated`. Variable `exact: real` is initialized with the exact value of the integral from Mathematica, 0.25. It then checks to see if the absolute value of the difference between `calculated` and `exact` is less than or equal to `maximumError` and sets `verified: bool`. The test writes out `verified` and a passing test results in **`true`**. R4.1

TODO. Refactor `chapelexample` environment to `chapeltest` environment or a more general `test` environment. (Similarly for `chapelsource` and `chapelhelper` environs.)

Test (leftRectangleIntegrationTest1.chpl). A test for `leftRectangleIntegration`.

```
use leftRightRectangleMaxErr;
use leftRectangleIntegration;

proc f(x:real):real {
  return x**3;
}

var calculated:real;
var exact:real = 0.25;  // from Mathematica
var maximumError:real = leftRightRectangleMaxErr(a = 0.0, b = 1.0, N
= 100, f = f);
var verified:bool;

calculated = leftRectangleIntegration(a = 0.0, b = 1.0, N = 100, f = f);
verified = (abs(calculated - exact) <= maximumError);
writeln(verified);
```

`seamless`. Now that we have our first test written, we need to extract it from the latex source and verify that it does not pass. To extract the test from the latex source and run it:

```
[./tutorial/] $ make tests
[./tutorial/] $ make test
```

These commands run the `tests` and `test` targets in the same Makefile referenced above. In this case, the `tests` target cleans out the `./tutorial/test` directory and executes the `./util/extract_tests` python script with the appropriate arguments. The `test` target changes to the `./tutorial/test` directory and executes the `start_test` csh script that comes with the chapel distribution (in `CHPL_HOME/util`). The script compiles and executes each of the chapel source files in the test directory (*e.g.* `leftRectangleIntegrationTest.chpl` as in the example above) and compares the output with the contents of a file with a `.good` extension (*e.g.* `leftRectangleIntegrationTest.good` for the above test). The last few lines of output should look something like this:

```
[Test Summary - 150107.202408]
[Summary: #Successes = 0 | #Failures = 1 | #Futures = 0 | #Warnings = 0 ]
[END]
```

TODO. Update test target to run all targets necessary to run tests.

The code that provides the `leftRectangleIntegration` function is straightforward.

S3 Function `leftRectangleIntegration`, for an interval of integration, $[a, b]$, takes the left end value of the interval, `a: real`, the right end value of the interval, `b: real`, the number of subintervals for the numerical integration, `N: int`, and the function to be integrated, `f`. The function stores the width of the subinterval calculated from Equation 5.2 in the variable `h: real`. It initializes the variable `sum: real` to zero, and for each value of n in the summation of Equation 5.1, it computes `x_n: real` according to the expression in Table 5.1 and adds the value of `f(x_n)` to `sum: real`. The function returns the product of `sum: real` and the subinterval width, `h: real`. R1.1, R2, R3

Source (leftRectangleIntegration.chpl).

```
proc leftRectangleIntegration(a: real(64), b: real(64), N: int(64), f): real(64){
  var h: real(64) = (b - a)/N;
  var sum: real(64) = 0.0;
  var x_n: real(64);
  for n in 0..N-1 {
    x_n = a + n * h;
    sum = sum + f(x_n);
  }
  return h * sum;
}
```

`seamless`. We can now verify that test `leftRectangleIntegrationTest1.chpl` passes. First we need to extract the chapel source from our latex file and then run the test that was written previously:

```
[./tutorial/] $ make sources
[./tutorial/] $ make test
```

These commands run the `sources` and `test` targets in our Makefile. In this case, the `sources` target cleans out the `./tutorial/source` directory and executes the `./util/extract_sources` python script with the appropriate arguments, putting the source code that we've defined in our latex file into the `./tutorial/source` directory. The last few lines of output should look something like this:

```
[Test Summary - 150107.202408]
[Summary: #Successes = 1 | #Failures = 0 | #Futures = 0 | #Warnings = 0 ]
[END]
```

Another of the functions that we need to test our methods against is $f(x) = 1/x$, where x is $[1, 100]$, with 1,000 approximations. The exact result is the natural log of 100, or about 4.605170. Since the function is decreasing on the interval $[1, 100]$, we can again use the helper function in `leftRightRectangleMaxErr.chpl` to compute the maximum expected error. Our second test for the left rectangle method is very similar to the first.

S4 Test `leftRectangleIntegrationTest2.chpl` loads modules `leftRightRectangleMaxErr` and `leftRectangleIntegration`. It defines a function `f` that takes `x: real` and returns $1/x$. It passes $a = 1.0$, $b = 100.0$, $N = 1000$, and `f` to the function `leftRightRectangleMaxErr` and stores the result in the variable `maximumError` (real). It passes $a = 1.0$, $b = 100.0$, $N = 1000$, and `f` to the function `leftRectangleIntegration` and stores the result in the variable `calculated`. Variable `exact: real` is initialized with the exact value of the integral, 4.605170. It then checks to see if the absolute value of the difference between `calculated` and `exact` is less than or equal to `maximumError` and sets `verified: bool`. The test writes out `verified` and a passing test results in `true`. R4.2

Test (leftRectangleIntegrationTest2.chpl). A test for `leftRectangleIntegration` using $f(x) = 1/x$.

```
use leftRightRectangleMaxErr;
use leftRectangleIntegration;

proc f(x:real):real {
  return 1/x;
}

var exact:real = 4.605170;
var maximumError:real = leftRightRectangleMaxErr(a = 1.0, b = 100.0, N
= 1000, f = f);
var calculated: real = leftRectangleIntegration(a = 1.0, b = 100.0, N = 1000, f = f);
var verified: bool = (abs(calculated - exact) <= maximumError);
writeln(verified);
```

`seamless`. By now you have likely realized that we already have some opportunities to refactor code in our first two tests above. The tests are very similar except for the expressions in the test function (f), the exact values for the integrals, and the values of a, b, and N passed to `leftRightRectangleMaxErr` and `leftRectangleIntegration`. Also, you'll notice that the arguments to `leftRightRectangleMaxErr` and `leftRectangleIntegration` are identical, and perhaps it would be good to always get the maximum error associated with a numerical integration. We will rewrite our integration function to return the value of the integral and the maximum error in a tuple. We can also combine the two tests and add the final two tests for the function $f(x) = x$. Thinking ahead a little, we can put the test functions into a helper module since they will be the same for every test. In practice, the developer would typically not keep the above two tests. She would replace the above two tests with what follows and the resulting `seamless` document would be much more streamlined than what is presented here. Of course, there is no harm in keeping all of the versions of the tests.

TODO. Add description of versioning with git.

Helper (testFunctions.chpl).

```
proc f1(x:real):real {
  return x**3;
}
proc f2(x:real):real {
  return 1/x;
}
proc f3(x:real):real {
  return x;
}
```

Test (leftRectangleIntegrationTest3.chpl). A test for `leftRectangleIntegration` using $f(x) = \{x^3, 1/x, x\}$.

```
use leftRectangleIntegrationWithErr;
use testFunctions;

var exact:real;
var calculated:real;
var maxErr:real;

exact = 0.25;
(maxErr, calculated) = leftRectangleIntegrationWithErr(a = 0.0, b = 1.0,
N = 100, f = f1);
writeln((abs(calculated - exact) <= maxErr));

exact = 4.605170;
(maxErr, calculated) = leftRectangleIntegrationWithErr(a = 1.0, b = 100.0,
N = 1000, f = f2);
writeln((abs(calculated - exact) <= maxErr));

exact = 12500000;
(maxErr, calculated) = leftRectangleIntegrationWithErr(a = 0.0, b = 5000.0,
N = 5000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));
```

```
    exact = 18000000;
    (maxErr, calculated) = leftRectangleIntegrationWithErr(a = 0.0, b = 6000.0,
N = 6000000, f = f3);
    writeln((abs(calculated - exact) <= maxErr));
```

The code that provides the `leftRectangleIntegrationWithErr` function is straightforward.

S5 Function `leftRectangleIntegrationWithErr`, for an interval of integration, $[a, b]$, takes the left end value of the interval, `a: real`, the right end value of the interval, `b: real`, the number of subintervals for the numerical integration, `N: int`, and the function to be integrated, `f`. The function stores the width of the subinterval calculated from Equation 5.2 in the variable `h: real`. It initializes the variable `sum: real` to zero, and for each value of n in the summation of Equation 5.1, it computes `x_n: real` according to the expression in Table 5.1 and adds the value of `f(x_n)` to `sum: real`. The function returns the product of `sum: real` and the subinterval width, `h: real` as the first element of a two-element tuple.. The second element of the returned tuple is the maximum error expected calculated according to equation 5.3. **??**, **??**, **??**

Source (leftRectangleIntegrationWithErr.chpl).

```
proc leftRectangleIntegrationWithErr(a: real(64), b: real(64), N: int(64), f): 2*real{
  var maxErr: real = ((b-a)/N)*abs(f(b)-f(a));
  var h: real(64) = (b - a)/N;
  var sum: real(64) = 0.0;
  var x_n: real(64);
  for n in 0..N-1 {
    x_n = a + n * h;
    sum = sum + f(x_n);
  }
  return (h * sum, maxErr);
}
```

5.2 Right Rectangle Method

Test (rightRectangleIntegrationTest.chpl). A test for `rightRectangleIntegrationWithErr` using $f(x) = \{x^3, 1/x, x\}$.

```
use rightRectangleIntegrationWithErr;
use testFunctions;

var exact:real;
var calculated:real;
var maxErr:real;

exact = 0.25;
(maxErr, calculated) = rightRectangleIntegrationWithErr(a = 0.0, b = 1.0,
N = 100, f = f1);
writeln((abs(calculated - exact) <= maxErr));

exact = 4.605170;
(maxErr, calculated) = rightRectangleIntegrationWithErr(a = 1.0, b = 100.0,
N = 1000, f = f2);
writeln((abs(calculated - exact) <= maxErr));
```

```
    exact = 12500000;
    (maxErr, calculated) = rightRectangleIntegrationWithErr(a = 0.0, b = 5000.0,
N = 5000000, f = f3);
    writeln((abs(calculated - exact) <= maxErr));

    exact = 18000000;
    (maxErr, calculated) = rightRectangleIntegrationWithErr(a = 0.0, b = 6000.0,
N = 6000000, f = f3);
    writeln((abs(calculated - exact) <= maxErr));
```

Source (rightRectangleIntegrationWithErr.chpl).

```
proc rightRectangleIntegrationWithErr(a: real(64), b: real(64), N: int(64), f): 2*real{
  var maxErr: real = ((b-a)/N)*abs(f(b)-f(a));
  var h: real(64) = (b - a)/N;
  var sum: real(64) = 0.0;
  var x_n: real(64);
  for n in 0..N-1 {
    x_n = a + (n + 1) * h;
    sum = sum + f(x_n);
  }
  return (h * sum, maxErr);
}
```

5.3 Midpoint Rectangle Method

For a function f which is twice differentiable, the maximum error E for the midpoint rectangle method is given by the following equation:

$$E \le \frac{(b-a)^3}{24N^2} f''(\xi) \tag{5.4}$$

for some ξ in $[a, b]$.

Unlike the left and right rectangle methods, it is very difficult to write a function to determine the maximum expected error. First, we must determine the maximum value of the second derivative before we can compute the maximum error using Equation 5.4. For $f(x) = x^3$, the second derivative is $f''(x) = 6x$. On the interval specified by Requirement R4.1, $[0, 1]$, the maximum value is $f''(1) = 6$. For $f(x) = 1/x$, the second derivative is $f''(x) = 2x^{-3}$. On the interval specified by Requirement R4.2, $[1, 100]$, the maximum value is $f''(1) = 2$. The function $f(x) = x$ specified by Requirement R4.3 and R4.4, does not have a second derivative. The midpoint method is expected to give a very accurate answer for this function, so we will use a value of 0.00001 for the maximum expected error for the two final tests. The calculated maximum expected error for the tests specified in Requirements R4.1 and R4.2 are given in Table 5.2.

Table 5.2: Values for expressions in Equation 5.4 and the maximum expected error of the midpoint rectangle method of numerical integration for $f(x) = \{x^3, 1/x\}$.

Function	**Interval**	**N**	**Maximum** $f''(x)$	E
x^3	$[0, 1]$	100	6	0.000025
$1/x$	$[1, 100]$	1000	3	0.121287

Test (midpointRectangleIntegrationTest.chpl). A test for `midpointRectangleIntegrationWithErr` using $f(x) = \{x^3, 1/x, x\}$.

```
    use midpointRectangleIntegrationWithErr;
    use testFunctions;

    var exact:real;
    var calculated:real;
    var maxErr:real;

    exact = 0.25;
    maxErr = 0.000025;
    calculated = midpointRectangleIntegrationWithErr(a = 0.0, b = 1.0, N = 100, f = f1);
    writeln((abs(calculated - exact) <= maxErr));

    exact = 4.605170;
    maxErr = 0.121287;
    calculated = midpointRectangleIntegrationWithErr(a = 1.0, b = 100.0, N
= 1000, f = f2);
    writeln((abs(calculated - exact) <= maxErr));

    exact = 12500000;
    maxErr = 0.00001;
    calculated = midpointRectangleIntegrationWithErr(a = 0.0, b = 5000.0, N
= 5000000, f = f3);
    writeln((abs(calculated - exact) <= maxErr));

    exact = 18000000;
    maxErr = 0.00001;
    calculated = midpointRectangleIntegrationWithErr(a = 0.0, b = 6000.0, N
= 6000000, f = f3);
    writeln((abs(calculated - exact) <= maxErr));
```

Source (midpointRectangleIntegrationWithErr.chpl).

```
proc midpointRectangleIntegrationWithErr(a: real(64), b: real(64), N: int(64), f): real{
  var h: real(64) = (b - a)/N;
  var sum: real(64) = 0.0;
  var x_n: real(64);
  for n in 0..N-1 {
    x_n = a + (n + 0.5) * h;
    sum = sum + f(x_n);
  }
  return h * sum;
}
```

6 Trapezoid Integration

The trapezoid method computes an approximation to a definite integral by finding the area of a collection of trapezoids whose heights are determined by the values of the function. Specifically, the interval $[a, b]$ over which the function is to be integrated is divided into N equal subintervals of length $h = (b - a)/N$. The trapezoids are drawn with the base along the x-axis. Both the left and right corner of the side opposite the base lies on the graph of the function. The approximation to the integral is then calculated by adding up the areas of the trapezoids (base multiplied by sum of two sides divided by two) of the N trapezoids, giving the formula:

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{n=1}^{N-1} f(x_n) + f(b)\right] \tag{6.1}$$

where

$$x_n = a + nh \tag{6.2}$$

and h is still given by Equation 5.2.

For a function f which is twice differentiable, the maximum error E for the trapezoid method is given by the following equation:

$$E \le \frac{(b-a)^3}{12N^2} f''(\xi) \tag{6.3}$$

for some ξ in $[a, b]$.

We can use the maximum value of the second derivative computed in Section 5.3 in Equation 6.3. As with the midpoint rectangle method, the trapezoid method is expected to give a very accurate answer for $f(x) = x$, so we will use a value of 0.00001 for the maximum expected error for the two final tests. The calculated maximum expected error for the tests specified in Requirements **??** and **??** are given in Table 6.1.

Table 6.1: Values for expressions in Equation 6.3 and the maximum expected error of the trapezoid method of numerical integration for $f(x) = \{x^3, 1/x\}$.

Function	**Interval**	**N**	**Maximum $f''(x)$**	E
x^3	$[0, 1]$	100	6	0.00005
$1/x$	$[1, 100]$	1000	3	0.24257

Test (trapezoidIntegrationTest.chpl). A test for `trapezoidIntegration` using $f(x) = \{x^3, 1/x, x\}$.

```
use trapezoidIntegration;
use testFunctions;

var exact:real;
var calculated:real;
var maxErr:real;

exact = 0.25;
maxErr = 0.00005;
calculated = trapezoidIntegration(a = 0.0, b = 1.0, N = 100, f = f1);
writeln((abs(calculated - exact) <= maxErr));
```

```
exact = 4.605170;
maxErr = 0.24257;
calculated = trapezoidIntegration(a = 1.0, b = 100.0, N = 1000, f = f2);
writeln((abs(calculated - exact) <= maxErr));

exact = 12500000;
maxErr = 0.00001;
calculated = trapezoidIntegration(a = 0.0, b = 5000.0, N = 5000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));

exact = 18000000;
maxErr = 0.00001;
calculated = trapezoidIntegration(a = 0.0, b = 6000.0, N = 6000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));
```

Source (trapezoidIntegration.chpl).

```
proc trapezoidIntegration(a: real(64), b: real(64), N: int(64), f): real{
  var h: real(64) = (b - a)/N;
  var sum: real(64) = f(a) + f(b);
  var x_n: real(64);
  for n in 1..N-1 {
    x_n = a + n * h;
    sum = sum + 2.0 * f(x_n);
  }
  return (h/2.0) * sum;
}
```

7 Simpson's Rule Integration

The Simpson's rule method approximates the function with a quadratic. The particular flavor that we are going to use here requires that the interval $[a, b]$ is subdivided into an even number of subintervals of width $h = (b - a)/N$. The general Simpson's rule is given by

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[f(a) + 4\sum_{\substack{n=1 \\ n\text{odd}}}^{N-1} f(x_n) + 4\sum_{\substack{n=2 \\ n\text{even}}}^{N-2} f(x_n) + f(b)\right] \tag{7.1}$$

where x_n is still given by Equation 6.2 and h is still given by Equation 5.2.

For a function f which has a fourth derivative, the maximum error E for the Simpson's rule method is given by the following equation:

$$E \leq \frac{(b-a)^5}{180N^4} f^{(4)}(\xi) \tag{7.2}$$

for some ξ in $[a, b]$.

The expected error for the Simpson's rule method for all of the tests is expected to be very low, so we will use a value of 0.00001 for all of them.

Test (simpsonsIntegrationTest.chpl). A test for `simpsonsIntegration` using $f(x) = \{x^3, 1/x, x\}$.

```
use simpsonsIntegration;
use testFunctions;

var exact:real;
var calculated:real;
var maxErr:real;

exact = 0.25;
maxErr = 0.00001;
calculated = simpsonsIntegration(a = 0.0, b = 1.0, N = 100, f = f1);
writeln((abs(calculated - exact) <= maxErr));

exact = 4.605170;
maxErr = 0.00001;
calculated = simpsonsIntegration(a = 1.0, b = 100.0, N = 1000, f = f2);
writeln((abs(calculated - exact) <= maxErr));

exact = 12500000;
maxErr = 0.00001;
calculated = simpsonsIntegration(a = 0.0, b = 5000.0, N = 5000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));

exact = 18000000;
maxErr = 0.00001;
calculated = simpsonsIntegration(a = 0.0, b = 6000.0, N = 6000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));
```

Source (simpsonsIntegration.chpl).

```
proc simpsonsIntegration(a: real(64), b: real(64), N: int(64), f): real{
  var h: real(64) = (b - a)/N;
  var sum: real(64) = f(a) + f(b);
  var x_n: real(64);
  for n in 1..N-1 by 2 {
    x_n = a + n * h;
    sum = sum + 4.0 * f(x_n);
  }
  for n in 2..N-2 by 2 {
    x_n = a + n * h;
    sum = sum + 2.0 * f(x_n);
  }
  return (h/3.0) * sum;
}
```

Test (simpsonsIntegrationParallelTest.chpl). A test for `simpsonsIntegrationParallel` using $f(x) = \{x^3, 1/x, x\}$.

```
use simpsonsIntegrationParallel;
use testFunctions;

var exact:real;
var calculated:real;
var maxErr:real;

exact = 0.25;
maxErr = 0.00001;
calculated = simpsonsIntegrationParallel(a = 0.0, b = 1.0, N = 100, f = f1);
writeln((abs(calculated - exact) <= maxErr));

exact = 4.605170;
maxErr = 0.00001;
calculated = simpsonsIntegrationParallel(a = 1.0, b = 100.0, N = 1000, f = f2);
writeln((abs(calculated - exact) <= maxErr));

exact = 12500000;
maxErr = 0.00001;
calculated = simpsonsIntegrationParallel(a = 0.0, b = 5000.0, N = 5000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));

exact = 18000000;
maxErr = 0.00001;
calculated = simpsonsIntegrationParallel(a = 0.0, b = 6000.0, N = 6000000, f = f3);
writeln((abs(calculated - exact) <= maxErr));
```

Source (simpsonsIntegrationParallel.chpl).

```
proc simpsonsIntegrationParallel(a: real(64), b: real(64), N: int(64), f): real{
  var h: real(64) = (b - a)/N;
  var sum1,sum2: sync real = 0.0;
  var x_n1,x_n2: sync real;
  cobegin {
    for n1 in 1..N-1 by 2 {
      x_n1=a+n1*h;sum1 = sum1+4.0*f(x_n1);
    }
    for n2 in 2..N-2 by 2 {
      x_n2=a+n2*h;sum2 = sum2+2.0*f(x_n2);
    }
  }
  return (h/3.0) * (f(a) + sum1+sum2 + f(b));
}
```

TODO. Fix chapel_listing.tex to handle $ characters for sync variables.

Test (simpsonsIntegrationParallelTestWithTiming.chpl). A test for `simpsonsIntegration` and `simpsonsIntegrationParallel` using $f(x) = \{x^3, 1/x, x\}$ comparing timing of the parallel and serial methods.

```
use simpsonsIntegrationParallel;
use simpsonsIntegration;
use testFunctions;
use Time;

var exact:real;
var calculated:real;
var maxErr:real;
var timer, timerP:Timer;
var timerMargin: real = 0.6;

exact = 0.25;
maxErr = 0.00001;
timerP.start();
calculated = simpsonsIntegrationParallel(a = 0.0, b = 1.0, N = 100, f = f1);
timerP.stop();
writeln((abs(calculated - exact) <= maxErr));
timer.start();
calculated = simpsonsIntegration(a = 0.0, b = 1.0, N = 100, f = f1);
timer.stop();
writeln((abs(calculated - exact) <= maxErr));
writeln((timerP.elapsed() < timerMargin*timer.elapsed()));

exact = 4.605170;
maxErr = 0.00001;
timerP.clear();
timerP.start();
calculated = simpsonsIntegrationParallel(a = 1.0, b = 100.0, N = 1000, f = f2);
timerP.stop();
writeln((abs(calculated - exact) <= maxErr));
timer.clear();
timer.start();
calculated = simpsonsIntegration(a = 1.0, b = 100.0, N = 1000, f = f2);
timer.stop();
writeln((abs(calculated - exact) <= maxErr));
writeln((timerP.elapsed() < 0.5*timer.elapsed()));

exact = 12500000;
maxErr = 0.00001;
timerP.clear();
timerP.start();
calculated = simpsonsIntegrationParallel(a = 0.0, b = 5000.0, N = 5000000, f = f3);
timerP.stop();
writeln((abs(calculated - exact) <= maxErr));
timer.clear();
timer.start();
calculated = simpsonsIntegration(a = 0.0, b = 5000.0, N = 5000000, f = f3);
timer.stop();
writeln((abs(calculated - exact) <= maxErr));
writeln((timerP.elapsed() < timerMargin*timer.elapsed()));

exact = 18000000;
maxErr = 0.00001;
timerP.clear();
timerP.start();
calculated = simpsonsIntegrationParallel(a = 0.0, b = 6000.0, N = 6000000, f = f3);
timerP.stop();
writeln((abs(calculated - exact) <= maxErr));
timer.clear();
timer.start();
calculated = simpsonsIntegration(a = 0.0, b = 6000.0, N = 6000000, f = f3);
timer.stop();
writeln((abs(calculated - exact) <= maxErr));
writeln((timerP.elapsed() < timerMargin*timer.elapsed()));
```

Table 1: Requirement traceability matrix.

Requirement	**Specification**
R1.1	**S3**
R1.2	
R1.3	
R1.4	
R1.5	
R2	**S3**
R3	**S3**
R4.1	**S2**
R4.2	**S4**
R4.3	
R4.4	

Index

arguments
    formal, 17

formal arguments, 17
functional requirements, 15

integration
    rectangle, 17
    Simpson's rule, 29
    trapezoid, 27

notation, 11

operator precedence, 17
operators
    precedence, 17
organization, 13

rectangle integration, 17
requirements, 15

scope, 15
Simpson's rule integration, 29

trapezoid integration, 27

Bibliography

[1] Numerical integration-rosetta code. http://rosettacode.org/wiki/Numerical_integration. Accessed: 2015-01-01.

[2] Bart Childs. *Literate Programming, A Practitioner's View*, pages 261–262. Tugboat, December 1992.

[3] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[4] Marian Petre and Greg Wilson. Plos/mozilla scientific code review pilot: Summary of findings. *CoRR*, abs/1311.2412, 2013.

[5] Ross N. Williams. Funnelweb tutorial manual: What is literate programming? http://www.ross.net/funnelweb/tutorial/intro_what.html. Accessed: 2014-01-02.