Test-Driven Development with `seamless`

Paul Adamson

January 1, 2015

Contents

1 Introduction

Several characteristics of legacy code present challenges to the scientific computing community. These characteristics are symptoms of the fact that, in general, the codes are written by individuals who are scentists or engineers first and computer programmers second. Often times, a code is very poorly documented, and the test suite amounts to a few exaample input decks and expected output decks. The vast majority of modules, procedures, and objects within a code will likely not have unit tests associated with them. In addition to basic shortfalls in documentation, there is often a lack of context in the documentation that is available. Although adaptation patterns can be found in a sizable percentage of recently developed science codes, the traditional scientific software stack lacks adequate abstractions and tools necessary to port it to a new execution environment or apply it to a problem set different than what was originally intended.[3] While scientists are incentivized and indeed desire to produce reusable, maintainable code, these deficiencies are roadblocks to that end.[5]

> *TODO*. Add paragraph on overhead of good software engineering. Integration of multiple tools.

The purpose of the `seamless` framework for science code development is to provide scientists and engineers with some tools to help them produce code that is reusable and maintainable. In the `seamless` framework, a test-driven development approach is combined with concepts from literate programming[4] 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 `seamless` note text boxes used only in the tutorial:

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

LaTeX source, shell commands, and other computer syntax is placed in the `verbatim` environment and is typeset in a typewriter font. Many examples in this tutorial use command-line commands. All command line examples use a Unix-style command line prompt (a dollar sign) with the directory in brackets preceding it, as follows:

```
[./tutorial/] $ echo 'hello, world!'
hello, world!
```

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

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 delimiters" 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, separated 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[6]

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.[5] Literate programming solves this problem, ensuring that context is created while the program is written.

1.3 The `seamless` Framework

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 process also supports several fundamental aspects of good software engineering.

TODO. Add better overview of `seamless`. See intro slides.

1.3.1 Writing Good Requirements

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.

1.3.2 The Code Development Process Within the `seamless` Framework

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]
   (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.

[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.

(d) Refactor and retest some more.

Repeat the above cycle until your code is complete. This process will help to ensure that your code will have the following characteristics:

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

1.4 Tutorial Structure

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`

To build the PDF of the tutorial, simply run `make`. The default target in the makefile uses `pdflatex` to compile the package:

```
[./tutorial/] $ make
...
Output written on Numerical_Integration.pdf (43 pages, 306114 bytes).
Transcript written on Numerical_Integration.log.
```

Adapting the associated LaTeX files for a new software project is straightforward. Once you've adapted the structure of the LaTeX package in the `.\tutorial` directory for your purposes, and you've written a decent requirement specification, you're ready to begin the process described in Section 1.3.2.

1.5 `seamless` Does Not Fully Implement Literate Programming

The `seamless` framework borrows from literate programming, but it is not a full implementation of its ideas. While the approach that I've described meets the intent of the requirements outlined in Section 1.2 above, it only fully implement one of the two main concepts of literate programming.[4] The first concept, already described at length, 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

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

5   // 55 lines of error handling code

7   readNuclei(reader);
8   readBasis(reader);

10 }
```

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.

When you program in a true literate programming system, you get to write code in any order that supports documentation of the code. 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. This aspect of literate programming is not implemented in the `seamless` framework. This is a conscious decision that was made in developing the framework in order to allow the TDD approach to drive the order in which test and source code is developed and presented to the reviewer. The compromise allows us to take advantage of many of the characteristics of literate programming while executing a TDD process.

2 Notation

2.1 Code Chunks

TODO. Add a note about labeling requirements and specifications. Also, a note about how requirements traceability is accomplished.

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;
}
```

2.2 Examples

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);
```

2.3 Text Boxes

Different color text boxes are used to highlight *TODO's, Notes, Rationales, Open Issues,* and *Futures* throughout the specification. Examples of these text boxes along with definitions for each of these terms is given below:

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 1.3.1, 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. An example requirements listing follows:

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}` will place the label **R1.1**).

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. Now that we have our requirements documented, it is time to enter into the repetitive `seamless` TDD cycles necessary to actually document, test, and develop our code. First, some suggestions:

- there should be example programs for every package feature
- 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
- explain the impact of the feature and how it is used when piecing together an application
- italicize a term when you are defining it and be sure to add it to the index (more on index creation below)

In our toy example, we are developing a few numerical integration methods, and we begin with a description of the rectangle method. What follows is a just normal LaTeX prose including a figure generated in Mathematica to illustrate the concept. It is a good idea to include any external scripts, programs, or spreadsheets used to generate figures for your documentation, and the Mathematica notebook for generating Figure 5.1 is included with the tutorial in the `./tutorial/Mathematica` directory.

The *rectangle integration 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

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

seamless. Notice that the term "rectangle integration method" above is italicized at its first use. If you look in the index at the end of the tutorial, you will see two different entries for the term. One is a plain entry for "rectangle integration method," and the other is a subentry for "rectangle" under the "integration method" entry. The index entries are created by inserting `\index` commands in the LaTeX source and running the `makeindex` command (included for you in the `Makefile`). The plain entry is inserted by simply entering `\index{rectangle integration method}` after

the first occurrence of the term. The subentry is inserted by using a `!` to annotate the subentry after the main entry (*e.g.* `\index{integration method!rectangle}`). Some things to consider with respect to index term creation:

- enter as many terms as possible into the index; it is better to err on the side of too many entries; you can cull excessive entries later
- double-billing is encouraged (*e.g.* `\index{formal arguments}`, `\index{arguments!formal}`)
- an acceptable exception to double-billing is when the entries are visually close (*e.g.* `\index{operator precedence}`, `\index{operators!precedence}`), 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 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.

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	$a + \left(n + \frac{1}{2}\right) h$

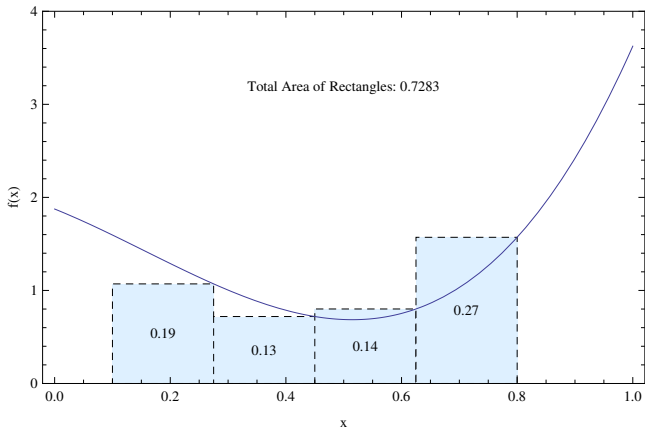
5.1 Left Rectangle Method

`seamless`. Now that we've described the rectangle method fairly well, including the math for each of the different versions, we will begin developing the code for the left rectangle method. The first step is to develop the tests, and every test needs an expected value against which to compare. The obvious way to test the code is to compare results against analytical values. For the tests specified in R4.1-R4.4, we have analytic values. Also, since the functions are increasing or decreasing over the intervals specified, we can use an expression for the maximum error of the rectangle integration method in our tests.

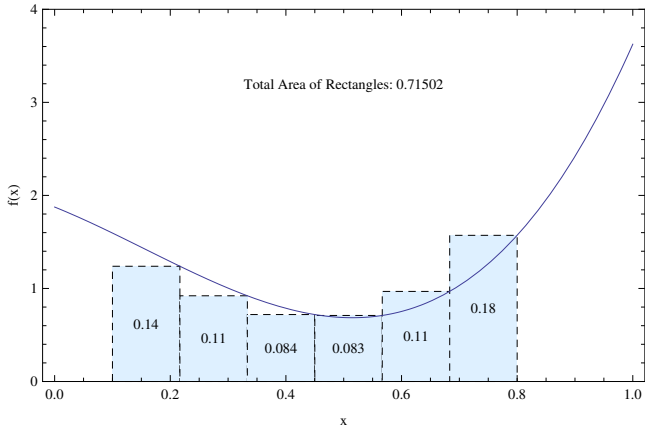
If $f(x)$ is increasing or decreasing on the interval $[a, b]$, the *maximum error*, E, for left or right rectangle method numerical integration is given by

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

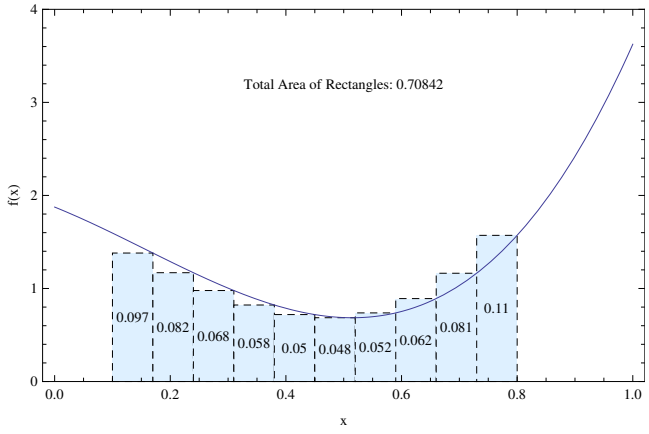
We can create a helper function to compute the maximum error for left and right rectangle methods using Equation 5.3. The calculated value will be used in tests for the left and right rectangle methods to check that the result is within the maximum error expected for a given a, b, and N.



(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.

S1 Helper function `leftRightRectangleMaxErr`: compute the maximum error expected for left or right rectangle method of numerical integration.

arguments	`a:`**`real`**	lower bound
	`b:`**`real`**	upper bound
	`N:`**`int`**	number of subintervals
	`f`	function
return	`:`**`real`**	maximum error expected for left or right rectangle method of numerical integration per Equation 5.3

Helper (leftRightRectangleMaxErr.chpl).

```
1 proc leftRightRectangleMaxErr(a: real, b: real, N: int, f): real{
2   return ((b-a)/N)*abs(f(b)-f(a));
3 }
```

seamless. Our first specification, **S1**, was created with the following LaTeX:

```
1  \begin{enumspec}
2  \item\spec{1} Helper function \chpl{leftRightRectangleMaxErr}:
3    compute the maximum error expected for left or right rectangle
4    method of numerical integration. \\
5    \begin{tabular}{r r p{10cm}} \toprule
6      \textbf{arguments} & \chpl{a:real} & lower bound \\
7                         & \chpl{b:real} & upper bound \\
8                         & \chpl{N:int}  & number of subintervals \\
9                         & \chpl{f}      & function \\ \midrule
10     \textbf{return}    & \chpl{:real}  & maximum error expected
11       for left or right rectangle method of numerical integration
12       per Equation~\ref{eq:lr-rectangle-max-error} \\ \bottomrule
13   \end{tabular}
14 \end{enumspec}
```

Each specification is essentially an `item` (see line 2) in an `enumspec` environment, (see lines 1 and 14) which is defined in the file `seamless.sty` and is an extension of the `enumerate` environment. These `item`'s are labeled using the `\spec{x}` command, where `x` is the desired number in the label (*e.g.* `\spec{27}` to get **S27**). The convention in `seamless` is that the specifications are numbered consecutively, and they are referenced with the command `\ref{spec@x}`. The appropriate format in which to present the code specification is the developer's choice. As seen in lines 5 through 13 above, the format used in this tutorial is to use the `tabular` environment, with an "argument" group and a "return" group. As will be seen below, when a specification meets a requirement, another row is added to annotate this.

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 the LaTeX source files, run the following command in the directory containing the 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 command-line 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`.

S2 Test `leftRectangleIntegrationTest1`: integrates the function x^3 over the interval $[0, 1]$ by the left rectangle numerical integration method with 100 subintervals and verifies that the result is within the expected maximum error of the exact value, 0.25.

output	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`leftRightRectangleMaxErr`	
	`leftRectangleIntegration`	
requirements met	R4.1	

`seamless`. The first test specification, **S2**, differs quite a bit from **S1**, which was for a helper function. This first test is an executable to be compiled and run without any input, and the output is written to `stdout`, Therefore, we have a row for the potential outputs, but not input, arguments, or a return value. We also load two modules in this test, so there is a row to annotate those. Finally, this test meets one of the requirements, R4.1, and that fact is annotated in the "requirements met" row. To annotate this, we use the command `\meetsreq{x}`, where `x` is the number of the requirement that is met (*e.g.* `\meetsreq{4.1}` to annotate that Requirement **R4.1** is met).

Test (leftRectangleIntegrationTest1.chpl). A test for function `leftRectangleIntegration` using $f(x) = x^3$.

```
1  use leftRightRectangleMaxErr;
2  use leftRectangleIntegration;

4  proc f(x:real):real {
5    return x**3;
6  }

8  var calculated:real;
9  var exact:real = 0.25;
10 var maximumError:real = leftRightRectangleMaxErr(
11   a = 0.0, b = 1.0, N = 100, f = f);
12 var verified:bool;

14 calculated = leftRectangleIntegration(
```

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

seamless. The above test is inserted with the following LaTeX code:

```
1  \begin{chapelexample}{leftRectangleIntegrationTest1.chpl}
2    A test for function \chpl{leftRectangleIntegration}.
3    \begin{chapelpre}
4    \end{chapelpre}
5    \begin{chapel}
6  use leftRightRectangleMaxErr;
7  use leftRectangleIntegration;
8
9  proc f(x:real):real {
10   return x**3;
11 }
12
13 var calculated:real;
14 var exact:real = 0.25;
15 var maximumError:real = leftRightRectangleMaxErr(
16   a = 0.0, b = 1.0, N = 100, f = f);
17 var verified:bool;
18
19 calculated = leftRectangleIntegration(
20   a = 0.0, b = 1.0, N = 100, f = f);
21 verified = (abs(calculated - exact) <= maximumError);
22 writeln(verified);
23   \end{chapel}
24   \begin{chapelpost}
25   \end{chapelpost}
26   \begin{chapeloutput}
27 true
28   \end{chapeloutput}
29  \end{chapelexample}
```

Tests are inserted within `chapeltest` environments (see lines 1 and 29). The option passed to the environment is the filename for the test. Each test is given a unique filename. The text following the `\begin{chapeltest}` command is printed in the PDF, but is discarded when the test is extracted from the LaTeX source. Inside the `chapeltest` environment, four other environments are declared:

- `chapelpre`: code inserted at beginning of the test but not printed in the PDF
- `chapel`: test code
- `chapelpost`: code inserted at end of test but not printed in the PDF
- `chapeloutput`: `stdout` output for a passed test; text is extracted to a file with the same base filename as the test with a `.good` extension

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` script that comes with the chapel distribution (in the `CHPL_HOME/util` directory). The script compiles and executes each of the chapel source files in the test directory (*e.g.* `leftRectangleIntegrationTest1.chpl` as in the example above) and compares the output with the contents of a file with a `.good` extension (*e.g.* `leftRectangleIntegrationTest1.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.

TODO. Refactor `chapeltest` environment to `test`. (Similarly for `chapelsource` and `chapelhelper` environments.)

TODO. Add option to gobble spaces on left side in `chapeltest` so that code can be indented in source.

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

S3 Function `leftRectangleIntegration`: integrates a function by the left rectangle method of numerical integration.

arguments	`a:`**`real`**	lower bound
	`b:`**`real`**	upper bound
	`N:`**`int`**	number of subintervals
	`f`	function
return	`:`**`real`**	definite integral computed by left rectangle method of numerical integration per Equations 5.1 and 5.2 and the appropriate expression for x_n in Table 5.1
requirements met	R1.1, R2, R3	

Source (leftRectangleIntegration.chpl).

```
1 proc leftRectangleIntegration(a: real, b: real, N: int, f): real{
2   var h: real = (b - a)/N;
3   var sum: real = 0.0;
4   var x_n: real;
5   for n in 0..N-1 {
6     x_n = a + n * h;
7     sum = sum + f(x_n);
8   }
9   return h * sum;
10 }
```

seamless. Our first chunk of source code is inserted with the following LaTeX source:

```
1  \begin{chapelsource}{leftRectangleIntegration.chpl}
2    \begin{chapel}
3      proc leftRectangleIntegration(
4        a: real, b: real, N: int, f): real{
5        var h: real = (b - a)/N;
6        var sum: real = 0.0;
7        var x_n: real;
8        for n in 0..N-1 {
9          x_n = a + n * h;
10         sum = sum + f(x_n);
11       }
12       return h * sum;
13     }
14   \end{chapel}
15 \end{chapelsource}
```

The source is specified in a `chapel` environment (see lines 2 and 14) within a `chapelsource` environment (see lines 1 and 15). The option passed to the `\begin{chapelsource}` command specifies the file in which the source is placed. Multiple chunks of source can be extracted into a given source file.

We can now verify that test `leftRectangleIntegrationTest1` 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 the 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 now 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`: integrates the function $1/x$ over the interval $[1, 100]$ by the left rectangle numerical integration method with 1000 subintervals and verifies that the result is within the expected maximum error of the exact value, $ln(100)$.

output	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`leftRightRectangleMaxErr` `leftRectangleIntegration`	
requirements met	R4.2	

Test (leftRectangleIntegrationTest2.chpl). A test for function `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.

S5 Helper `testFunctions.chpl`: provide the functions $f(x) \in \{x^3, 1/x, x\}$ for use in tests.

proc f1		
arguments	`x:real`	x
return	`:real`	x^3
proc f2		
arguments	`x:real`	x
return	`:real`	$1/x$
proc f3		
arguments	`x:real`	x
return	`:real`	x

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;
}
```

S6 Test `leftRectangleIntegrationTest3`: integrates test functions according to Requirements R4.1 through R4.4.

output x 4	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`testFunctions`	
	`leftRectangleIntegrationWithErr`	
requirements met	R4.1, R4.2, R4.3, R4.4	

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

```
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));
```

S7 Function `leftRectangleIntegrationWithErr`: computes the definite integral of a function by the left rectangle method of numerical integration. Also computes the maximum expected error.

arguments	`a:real`	lower bound
	`b:real`	upper bound
	`N:int`	number of subintervals
	`f`	function
return	`:real`	definite integral computed by left rectangle method of numerical integration per Equations 5.1 and 5.2 and the appropriate expression for x_n in Table 5.1 is first element of two-element tuple
	`:real`	maximum error expected per Equation 5.3 is second element of two-element tuple
requirements met	R1.1, R2, R3	

Source (leftRectangleIntegrationWithErr.chpl).

```
proc leftRectangleIntegrationWithErr(
  a: real, b: real, N: int, f): 2*real{
  var maxErr: real = ((b-a)/N)*abs(f(b)-f(a));
  var h: real = (b - a)/N;
  var sum: real = 0.0;
  var x_n: real;
  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

S8 Test `rightRectangleIntegrationTest`: integrates test functions according to Requirements R4.1 through R4.4 using the right rectangle numerical integration method and compares results to expected maximum error.

output x 4	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`testFunctions`	
	`rightRectangleIntegrationWithErr`	
requirements met	R4.1, R4.2, R4.3, R4.4	

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

```
1  use rightRectangleIntegrationWithErr;
2  use testFunctions;

4  var exact:real;
5  var calculated:real;
6  var maxErr:real;

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

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

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

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

S9 Function `rightRectangleIntegrationWithErr`: computes the definite integral of a function by the right rectangle method of numerical integration. Also computes the maximum expected error.

arguments	`a:real`	lower bound
	`b:real`	upper bound
	`N:int`	number of subintervals
	`f`	function
return	`:real`	definite integral computed by right rectangle method of numerical integration per Equations 5.1 and 5.2 and the appropriate expression for x_n in Table 5.1 is first element of two-element tuple
	`:real`	maximum error expected per Equation 5.3 is second element of two-element tuple
requirements met	R1.2, R2, R3	

Source (rightRectangleIntegrationWithErr.chpl).

```
proc rightRectangleIntegrationWithErr(
  a: real, b: real, N: int, f): 2*real{
  var maxErr: real = ((b-a)/N)*abs(f(b)-f(a));
  var h: real = (b - a)/N;
  var sum: real = 0.0;
  var x_n: real;
  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:

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

for some ξ in $[a, b]$.

Unlike the left and right rectangle methods, it is more difficult to write a function to determine the maximum expected error for the midpoint version. 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.

S10 Test `midpointRectangleIntegrationTest`: integrates test functions according to Requirements R4.1 through R4.4 using the midpoint rectangle numerical integration method and compares results to the

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

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

maximum expected error.

output x 4	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`testFunctions`	
	`midpointRectangleIntegration`	
requirements met	R4.1, R4.2, R4.3, R4.4	

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

```
1  use midpointRectangleIntegration;
2  use testFunctions;

4  var exact:real;
5  var calculated:real;
6  var maxErr:real;

8  exact = 0.25;
9  maxErr = 0.000025;
10 calculated = midpointRectangleIntegration(
11   a = 0.0, b = 1.0, N = 100, f = f1);
12 writeln((abs(calculated - exact) <= maxErr));

14 exact = 4.605170;
15 maxErr = 0.121287;
16 calculated = midpointRectangleIntegration(
17   a = 1.0, b = 100.0, N = 1000, f = f2);
18 writeln((abs(calculated - exact) <= maxErr));

20 exact = 12500000;
21 maxErr = 0.00001;
22 calculated = midpointRectangleIntegration(
23   a = 0.0, b = 5000.0, N = 5000000, f = f3);
24 writeln((abs(calculated - exact) <= maxErr));

26 exact = 18000000;
27 maxErr = 0.00001;
28 calculated = midpointRectangleIntegration(
29   a = 0.0, b = 6000.0, N = 6000000, f = f3);
30 writeln((abs(calculated - exact) <= maxErr));
```

S11 Function `midpointRectangleIntegration`: computes the definite integral of a function by the midpoint rectangle method of numerical integration.

arguments	`a:real`	lower bound
	`b:real`	upper bound
	`N:int`	number of subintervals
	`f`	function
return	`:real`	definite integral computed by midpoint rectangle method of numerical integration per Equations 5.1 and 5.2 and the appropriate expression for x_n in Table 5.1
requirements met	R1.3, R2, R3	

Source (midpointRectangleIntegration.chpl).

```
1  proc midpointRectangleIntegration(
2    a: real, b: real, N: int, f): real{
3    var h: real = (b - a)/N;
4    var sum: real = 0.0;
5    var x_n: real;
6    for n in 0..N-1 {
7      x_n = a + (n + 0.5) * h;
8      sum = sum + f(x_n);
9    }
10   return h * sum;
11 }
```

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:

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

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

S12 Test `trapezoidIntegrationTest`: integrates test functions according to Requirements R4.1 through R4.4 using the trapezoid numerical integration method and compares results to the maximum expected error.

output x 4	`stdout: true` test passed `stdout: false` test failed
modules loaded	`testFunctions` `trapezoidIntegration`
requirements met	R4.1, R4.2, R4.3, R4.4

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));
```

S13 Function `trapezoidIntegration`: computes the definite integral of a function by the trapezoid method of numerical integration.

arguments	`a:real`	lower bound
	`b:real`	upper bound
	`N:int`	number of subintervals
	`f`	function
return	`:real`	definite integral computed by trapezoid method of numerical integration per Equations 6.1 and 5.2 and the expression for x_n in 6.2
requirements met	R1.4, R2, R3	

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.

S14 Test `simpsonsIntegrationTest.chpl`: integrates test functions using Simpson's method according to Requirements R4.1 through R4.4.

output x 4	`stdout: true`	test passed
	`stdout: false`	test failed
modules loaded	`testFunctions`	
	`simpsonsIntegration`	
requirements met	R4.1, R4.2, R4.3, R4.4	

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

```
1  use simpsonsIntegration;
2  use testFunctions;

4  var exact:real;
5  var calculated:real;
6  var maxErr:real;

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

14 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));
```

S15 Function `simpsonsIntegration`: computes the definite integral of a function by Simpson's method of numerical integration.

arguments	`a:`**`real`**	lower bound
	`b:`**`real`**	upper bound
	`N:`**`int`**	number of subintervals (must be even)
	`f`	function
return	**`:real`**	definite integral computed by Simpson's method of numerical integration per Equations 7.1 and 5.2 and the expression for x_n in 6.2
requirements met	R1.5, R2, R3	

TODO. Add check that N is even.

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;
}
```

A Requirements Traceability Matrix

Table A.1: Requirement traceability matrix.

Requirement	Specification
R1.1	**S3**, **S7**
R1.2	**S9**
R1.3	**S11**
R1.4	**S13**
R1.5	**S15**
R2	**S3**, **S7**, **S9**, **S11**, **S13**, **S15**
R3	**S3**, **S7**, **S9**, **S11**, **S13**, **S15**
R4.1	**S2**, **S6**, **S8**, **S10**, **S12**, **S14**
R4.2	**S4**, **S6**, **S8**, **S10**, **S12**, **S14**
R4.3	**S6**, **S8**, **S10**, **S12**, **S14**
R4.4	**S6**, **S8**, **S10**, **S12**, **S14**

Index

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] Pilsung Kang, Eli Tilevich, Srinidhi Varadarajan, and Naren Ramakrishnan. Maintainable and reusable scientific software adaptation: Democratizing scientific software adaptation. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 165–176, New York, NY, USA, 2011. ACM.

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

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

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