A Demonstration of the `seamless` Package: Numerical Integration in Chapel
Version 0.1

Paul Adamson

January 1, 2015

Contents

Part I

Introduction

1 Notation

Special notations are used in this specification to denote code. Code is represented with a fixed-width font where keywords are bold and comments are italicized.

Example.

```
for i in D do   // iterate over domain D
  writeln(i);   // output indices in D
```

2 Organization

This specification is organized as follows:

Part I Introduction

Chapter 1 Notation, introduces the notation that is used throughout this document.

Chapter 2 Organization, describes the contents of each of the parts and chapters within this document.

Chapter 3 Development Approach, describes the test-driven development process within the literate programming approach.

Part II Requirements Specification

Chapter 4 Scope, describes the scope of the package.

Chapter 5 Functional Requirements, describes the functional requirements of the package.

Part III Technical Specification

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

3 Development Approach

Before we dive into developing requirements and hacking away at code, a brief description of the `seamless` approach to software development, a literate programming approach to test-driven development, is in order. (Well, as you will see later, the `seamless` approach is probably better described as a "quasi-literate programming" approach, but I will explain in due course.)

3.1 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.

> *TODO:* Consider adding a story about TDD.

3.1.1 The Classic TDD Process

> *TODO:* The following process is almost ver batim from Rails 4 Test Prescriptions. Need to cite the work and tailor to technical computing/Chapel code development.

The classic TDD process goes something like this:

1. Create a test. The test should be short and test for one thing in your code. The test should run automatically.
2. Make sure the test fails. Verifying the test failure before you write code helps ensure that the test really does what you expect.
3. Write the simplest code that could possibly make the test pass. Don't worry about good code yet. Don't look ahead. Sometimes, write just enough code to clear the current error.
4. After the test passes, refactor to improve the code. Clean up duplication. Optimize. Create new abstractions. Refactoring is a key part of design, so don't skip this.
5. Run the tests again to make sure you haven't changed any behavior.

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.

3.1.2 TDD Aids Design

TODO: Describe in more detail how TDD aids design. Draw from Rails Test Prescriptions, pg 5+.

3.1.3 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!

3.2 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.

3.3 Literate Programming Approach to Test-Driven Development

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 scientific 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.

3.3.1 A Better TDD Process

A better TDD process begins first with a "good" requirement specification. Failing to write a specification 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 specification will always result in more time and lower quality code. Even for trivial examples, a short, informal specification will at least help to ensure accuracy of the resulting code.

The specification is the high-level design of the program. Most importantly, it clearly defines the problem that the program will solve. Of almost equal importance is the specification of the basic algorithms and outputs

of the code. During development of the requirement specification, the developer should evaluate available algorithms and consider how data produced from the program will be used. Even if a spec is written solely for the benefit of a lone developer, the act of writing the specification—describing how the program works in minute detail—will force design of the program.

Once a specification is in hand, an improved TDD process (section 3.1.1) can be undertaken in context of literate programming:

1. Document 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.

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] 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.

3.3.2 Additional Software Engineering Considerations

TODO: Insert description of how above approach supports good software engineering (feedback to requirements, etc.).

3.4 `seamless` Package

The `seamless` package aims to enable a literate programming approach to test-driven development of Chapel code. It extends slightly functionality provided in the distribution of the Chapel language source for extracting test code from the Chapel language specification. The following files are provided in `seamless`:

/Makefile main project Makefile

/spec directory containing the LaTeX source for this document, including an example of the `seamless` approach to developing a numerical integration code in the Chapel language

/spec/Makefile the Makefile to build this document

/spec/spec.tex the main LaTeX file for this document; other LaTeX files not listed here are self-explanatory

/spec/Numerical_Integration.tex the chapter of this document that contains the example of a literate programming approach to test-driven development of chapel code

/spec/chapel_listing.tex used by the LaTeX `listing` package to prettyprint Chapel code

/spec/chapel_testing.tex defines environments for adding extra information about test code chunks

/spec/collect_syntax.pl Perl script that collects any syntax definitions and builds the Syntax.tex file

/spec/syntax_listing.tex used by the LaTeX `listing` package to prettyprint Chapel syntax

/util/extract_tests Python script that extracts test code from LaTeX source

/util/extract_sources Python script that extracts source code from LaTeX source

TODO: combine extract python scripts and update above list

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 3.3.1. To illustrate the process, we will solve the Rosetta Code numerical integration task[1] in Chapel. As I go through the example, I will highlight how to use the `seamless` package to execute the literate programming and test-driven development approach.

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 3.2 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.

Part II

Requirements Specification

4 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.

5 Functional Requirements

Future. Consider creating a `requirements` LaTeXpackage to facilitate creating/cross-referencing lists of requirements and generating a requirement verification matrix...and other requirement enginering processes.

R1 The code shall have functions to calculate the definite integral of a function ($f(x)$).

R2 Available methods of integration shall include:

- **R2.1** rectangular
  - **R2.1.1** left
  - **R2.1.2** right
  - **R2.1.3** midpoint
- **R2.2** trapezium
- **R2.3** Simpson's

R3 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).

R4 The integration functions shall return the value for the integral.

R5 The test suite shall demonstrate the code's capability by showing the results for the following cases:

- **R5.1** $f(x) = x^3$, where x is $[0, 1]$, with 100 approximations. The exact result is 1/4, or 0.25.
- **R5.2** $f(x) = 1/x$, where x is $[1, 100]$, with 1,000 approximations. The exact result is the natural log of 100, or about 4.605170.
- **R5.3** $f(x) = x$, where x is $[0, 5000]$, with 5,000,000 approximations. The exact result is 12,500,000.
- **R5.4** $f(x) = x$, where x is $[0, 6000]$, with 6,000,000 approximations. The exact result is 18,000,000.

Part III

Technical Specification

6 Numerical Integration

6.1 Rectangle Method

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:

$$\int_a^b f(x)dx \approx h \sum_{n=0}^{N-1} f(x_n) \tag{6.1}$$

where

$$h = (b - a)/N \tag{6.2}$$

The formula for x_n for the left, right, and midpoint methods are given in Table 6.1. As N gets larger, the rectangle method becomes more accurate. This is illustrated in the series of plots in Figure 6.1.

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

$$E \leq \frac{b - a}{N} |f(b) - f(a)| \tag{6.3}$$

S1 Create a helper function to compute the maximum error: R5

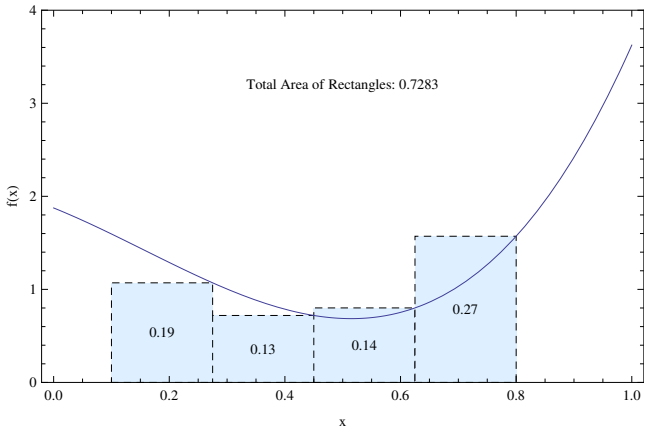
Example (leftRightRectangleIntegrationMaximumError.chpl).

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

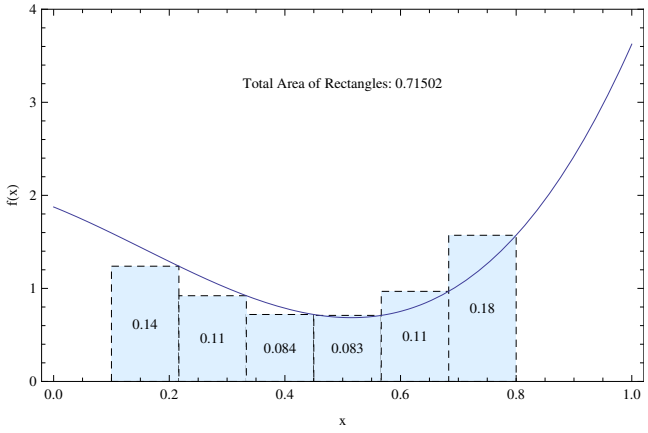
r6.5cm

Table 6.1: Formula for x_n in Equation 6.1 of rectangle numerical integration methods.

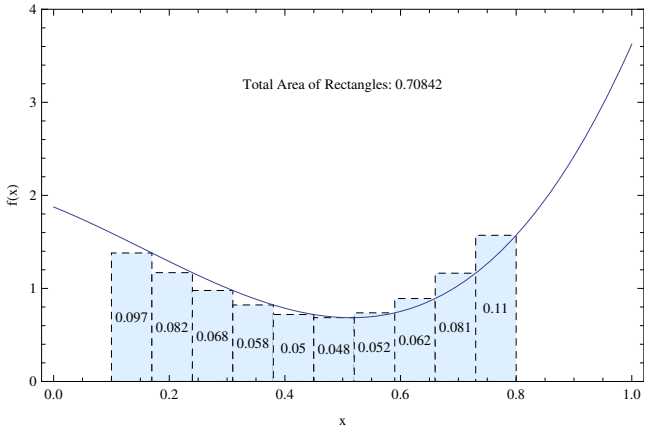
Rectangle Method	x_n
left	$a + nh$
right	$a + (n + 1)h$
midpoint	$a + \left(n + \frac{1}{2}\right) h$



(a) $N = 4$



(b) $N = 6$



(c) $N = 10$

Figure 6.1: 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 exact value of the integral is 0.70525.

S1 Testing for $f(x) = x^3$, with $a = 0$, $b = 1$, and $N = 100$. Since the function is increasing on the interval $[0, 1]$, the maximum error given by equation 6.3 is 0.01. R5.1

Example (leftRectangleIntegrationTest.chpl). A test for `leftRectangleIntegration`:

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

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

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

enter some text here...

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

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: real(64);
  for n in 0..N-1 {
    x = a + n * (b-a)/N;
    sum = sum + f(x);
  }
  return h * sum;
}
```

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

$$E \leq \frac{(b-a)h^2}{24} f''(\xi) \tag{6.4}$$

for some ξ in $[a, b]$.

Create a helper function to compute the maximum error:

Example (midpointRectangularIntegrationMaximumError.chpl).

```
proc midpointRectangularIntegrationMaximumError(a: real, b: real, N: int, fppxi){
  var h:real = (b-a)/N;
  return ((b-a)*h**2/24) * fppxi;
}
```

Table 2: Requirement traceability matrix.

Requirement	Specification
R1	
R2	
R2.1	
R2.1.1	
R2.1.2	
R2.1.3	
R2.2	
R2.3	
R3	
R4	
R5	**S1**
R5.1	**S1**
R5.2	
R5.3	
R5.4	

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] 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.