

# Appreciating functional programming: A beginner's tutorial to HASKELL illustrated with applications in numerical methods

*Chu Wei Lim*

`chuwei.lim@aostudies.com.sg`

*Weng Kin Ho \**

`wengkin.ho@nie.edu.sg`

National Institute of Education

Nanyang Technological University

637616

Singapore

## Abstract

*This paper introduces functional programming to the numerical methods community with the aim of popularizing this programming paradigm through a deeper appreciation for function as a mathematical concept and, at the same time, for its practical benefits. The functional language HASKELL is chosen amongst several choices because of its lazy evaluation strategy and high-performance compiler WinGHCi. We demonstrate the elegance and versatility of HASKELL by coding HASKELL programs to implement well-known numerical methods.*

## 1 Introduction

*Functional programming* is a style of programming which is an alternative to *imperative programming*; the latter being more commonly adopted in the programming community. More than just a stylistic difference, coding in a functional programming requires the programmer to put on a different mind-set. For this reason, we often refer to this new mind-set as the functional programming *paradigm*. No thinking occurs in vacuum. In line with the aims of the eJMT to focus on “all technology-based issues in all Mathematical Sciences”, we introduce functional programming (the *technology* part) in close relation to numerical methods (the *mathematics* part). The main purpose of this paper is to promote functional programming paradigm to mathematicians in this community as

---

\*Corresponding author

```

clear
n = input('Key in n: ');
value = 0;
% initialise value to 0
for i = 1:n
    value = value + i;
end
fprintf('Sum is %d. \n', value);

```

```

consecsum :: Int -> Int
consecsum 1 = 1
consecsum n = n + consecsum (n-1)

```

Figure 1: Programming styles: imperative (left) vs functional (right)

a novel way of thinking about numerical solutions of old problems. As a pleasant side effect, it is hoped that we have created here sufficient scenarios for tertiary mathematics students to explore and deepen their learning of mathematics (in the case, numerical methods) via functional programming. For this reason, our target audience would be mathematicians (and their students) who are familiar with *both* elementary numerical methods and at least one (imperative) programming language, such as MATLAB or C++.

For a quick taste of the paradigmatic difference between imperative and functional programming, let us consider the task of computing

$$\sum_{i=1}^n i = 1 + 2 + 3 + \cdots + n.$$

Figure 1 shows how the above summation is computed in imperative style (left), and in functional style (right).

With the imperative approach, a developer writes a code that specifies the steps which the computer must take to complete the task; this often is referred to as algorithmic programming. Because of the step-by-step specification style, there is a need to track this step-to-step transition using changes in state. In the imperative program (on the left), the change in state is enacted by an increment in the counter `i`. This change in state then results in a corresponding update in the variable `value`. We say that the variable `value` is *mutable* because the updated value is stored in the same register `value` after a state change occurs in `i`.

The flow control of an imperative-style program is typically initiated by loops (e.g., for-loops `for i = 1:n ... end`, and while-loops `while (conditional) do ...`), conditionals (e.g., `if ... then ... else`), and method calls. Table 1 shows the corresponding updates in `value` in each change in state for the input `n` equals to 4.

In contrast, a functional approach involves writing the program in the form of a set of pure mathematical functions to be executed. A *pure function* (or simply, function) is just an assignment of a *unique* output to each given input. A functional programmer focuses on what information is desired and what transformations are required, and this type of programming can be said to be declarative, i.e., the programmer declares what the function is to expect as the input and what to return as the output via some assignment rule. For example, in Figure 1 the program `consecsum` is of function type

i	value	Remarks
0	0	Initialize i
1	1	Start of for-loop
2	1 + 2	
3	1 + 2 + 3	
4	1 + 2 + 3 + 4	End of for-loop

Table 1: State changes and updates of `value`

```
consecsum :: Int -> Int
```

which expects to take in an integer (of type `Int`, naturally) and designed to output an integer.

Crucially, the functional approach does not make use of state changes to perform updates but instead exploits transformation of expressions to manipulate data. This can be seen as the execution of rewriting rules in a rewriting system. For instance, when `consecsum` operates on the input 4, the functional code (on the right) in Figure 1 instructs the computer to perform the following expression transformations:

```
consecsum 4 = 4 + consecsum 3
            = 4 + 3 + consecsum 2
            = 4 + 3 + 2 + consecsum 1
            = 4 + 3 + 2 + 1
```

The alert reader would have noticed the different roles of the equality symbol “=” in the clauses “`value = value + i`” (imperative) and “`consecsum n = n + consecsum (n-1)`” (functional), where the former stands for an assignment of an updated datum (previous datum added to the state number `i`) to the variable `value` while the latter defines a expression transformation that unfolds “`consecsum n`”.

In a nutshell, we see firstly that functional programming uses pure functions that return the same result if given the same arguments, i.e., pure functions are deterministic. Secondly all variables are *immutable*, that is, you cannot change the value of the variables. In other words the state of an object cannot change after it is created. The only way to effect any changes is to create a new object with a new value. The immediate benefits of functional programming is that every function is isolated and cannot impact other parts of the system. The deterministic nature of functions make them stable, consistent and predictable because we do not need to worry about situations in which the same variable can have different values.

In our ensuing development, we shall use relevant examples to bring out the various advantages of functional programming as a programming paradigm, and to demonstrate how functional programming helps us reinforce mathematical understanding. For a detailed computer-scientific comparison of functional and imperative programming paradigms, we refer the reader to the website [8].

The functional language we use here is HASKELL in which syntactic representation of programs bears strong resemblance to that of a function. More precisely, the code which implements a functional program  $f$  that takes in as input element from the data set  $A$  and returns an output element in the data set  $B$  always begins with the ‘domain-codomain’ kind of declaration:

$f :: A \rightarrow B$

Drawing on the reader’s familiarity with functions, we argue that the cognitive overhead for acquiring a functional programming language such as HASKELL is relatively lighter than an imperative one.

The bulk of this paper splits into two sections. Section 2 gives a quick tutorial on HASKELL, where many relevant examples will be brought in to illustrate the special features/advantages of functional programming. Section 3 illustrates how functional programming gives us a new way of thinking about mathematics by implementing elementary numerical methods in HASKELL. The way this paper is organized is for the convenience of readers with varying background knowledge. Readers who have some familiarity with functional programming and are only interested in HASKELL implementation of numerical methods may skip Section 2 and go directly to Section 3; perhaps referring to parts of Section 2 if the need arises. In the ensuing development, we often use the term *mathematician (HASKELL) programmer* to refer to mathematicians who write HASKELL programs to realize the mathematical functions they have in mind.

As this paper is meant to be an introductory tutorial, it is far from being comprehensive. For a detailed introduction to HASKELL, we refer the reader to [3] and [7], and for a discussion which is focused more on the functional programming paradigm itself, [4]. The website *Learn You a Haskell for Great Good!* (<http://learnyouahaskell.com/chapters>) is also a fun way of picking up HASKELL. We have also included herein an appendix “Getting started with HASKELL” that gives a quick guide on how to (1) install GHCi (the Glasgow HASKELL Compiler’s interactive environment), and (2) edit and compile HASKELL scripts (see A).

## 2 Functional programming with HASKELL

### 2.1 What’s HASKELL?

HASKELL<sup>1</sup> is a *purely functional* programming language, and so all computations are realized by applying (pure) functions on data/expressions to transform them. Every datum/expression can be assigned a (data) *type*, and types may be perceived as sets whose inhabitants are data/expressions.

Because of this perception, we want to think of HASKELL as a machine environment for us to create (or rather, re-create) mathematics from its very foundation. One foundational theory for mathematics is N ave Set Theory, where the primitive concept is ‘set’. Roughly speaking, more complicated sets are built from basic sets using set-theoretic axioms. We shall highlight the salient features of HASKELL by adopting the approach of building a mathematical universe within HASKELL – bottom up.

### 2.2 Values, expressions and types

At the lowest level, there are data which are printable, i.e., can be displayed on the screen or printed out. Printable data are called *values*, and the types that store these values are called *ground types*. In this paper, we only use the following ground types:

---

<sup>1</sup>The programming language HASKELL is named after Haskell Brooks Curry (1900–1989), an American mathematician and logician.

1. `Int`, the integer type contain the integers  $\dots, -2, -1, 0, 1, 2, \dots$ ;
2. `Bool`, the boolean type contain `true` and `false`;
3. `Double`, the double precision floating point number type.

Well-formed HASKELL programs are those which can be assigned a unique type, and this assignment is called a *typing assignment*, or simply, *typing*. Some instances of typing for values given below:

```
1 :: Int      true :: Bool      0.1 :: Double
```

In each of the above instances, “`::`” reads as “is of type”. The mathematician programmer may interpret `1 :: Int` as  $1 \in \mathbb{Z}$ . Of course, expressions in general can be assigned their types once the constituent terms in these expressions have been assigned their respective types. For instance, if we have the typing assignment

```
n :: Int
```

then the expression `n + 1` can be assigned the valid type:

```
n + 1 :: Int
```

Another example is the conditional “`if ... then ... else ...`”. If we assign the following types to the variables `c`, `t1`, `t2`

```
c :: Bool      t1, t2 :: A
```

then the expression `if c then t1 else t2` can be typed as

```
if c then t1 else t2 :: A
```

HASKELL enforces a set of typing rules to check whether expressions are correctly typed. Such a typing discipline forbids ill-formed programs such as

```
if 5 then 7 else 8
```

since it would have required that the datum `5` be typed as `Bool` but this is impossible because `5 :: Int` is the only legitimate type assignment for the integer `5`. We shall see later how this typing discipline guides the mathematician programmer to script his/her mathematical functions as legitimate HASKELL programs (see Example 10).

## 2.3 Building types

In N ave Set Theory, one would build more complicated sets from elementary ones using given set-theoretic axioms. For types, we can also build new types out of basic ground types, and to build more complex types out of these constructed types. In this paper, we only consider a small fragment of HASKELL where we work with the data types which are built from the basic ground types using a finite number of formation rules presented in the *Backus-Naur form* (BNF, for short) below:

$$A ::= \text{Int} \mid \text{Bool} \mid \text{Double} \mid (A, A) \mid A \rightarrow A \mid [A]$$

What the above BNF means is that a data type  $A$  is exactly one of these:

1. a ground type (i.e.,  $\text{Int}, \text{Bool}, \text{Double}$ ), or
2. a *product* type (i.e.,  $(A, A)$ ), or
3. a *function* type (i.e.,  $A \rightarrow A$ ), or
4. a *list* type (i.e.,  $[A]$ ).

**Remark 1** *A mathematician programmer may find it hard to wrap his/her head around the BNF-style of (inductive) type definition because  $A$  appears on both sides of the assignment  $::=$  and also in multiple occurrences, e.g.,  $A \rightarrow A$ . Do note that BNF is just a notation of the grammar for type formation and so  $::=$  is not an equation of any sort. Additionally, in the expression  $A \rightarrow A$  there is no obligation for the first instance of type  $A$  to be identical to the second instance of type  $A$ .*

We now explain which values belong to the product, the function and the list types respectively in the next subsections.

### 2.3.1 Product type

The type constructor  $(-, -)$  takes a pair of data types  $A$  and  $B$  to form the *product type*  $(A, B)$ . This is HASKELL's analogue of the set-theoretic Cartesian product,  $A \times B$ , of two sets  $A$  and  $B$ .

**Example 2 (Product type)** *Let us take the product type  $(\text{Int}, \text{Bool})$  which contains as data ordered pairs whose first component is of type  $\text{Int}$  and whose second component of type  $\text{Bool}$  as an example. A typical expression of the product type  $(\text{Int}, \text{Bool})$  is given by the ordered pair:*

$$(2, \text{True}) :: (\text{Int}, \text{Bool})$$

*Yet another example of a product type is  $(\text{Int}, \text{Int})$  which contains ordered pairs of integers.*

### 2.3.2 Function type

Central to the functional programming languages is the facility to produce function types. More precisely, given two data types  $A$  and  $B$  one forms the *function type*

$$A \rightarrow B.$$

The data which live in the function type  $A \rightarrow B$  are programs that take in an input of type  $A$  and return an output of type  $B$ . The function type constructor  $\rightarrow$  plays the same role in the universe of types as the function constructor  $(- \rightarrow -)$  does in the universe of sets; recall that given two sets  $A$  and  $B$ , then the collection of all the functions from  $A$  to  $B$ , denoted by

$$(A \rightarrow B)$$

is also a set by the axioms of Set Theory.

**Example 3 (Function type)** For a basic example, consider the first projection map `fst` that takes a data pair,  $(x, y)$ , of type  $(X, Y)$  and returns the first component  $x$  of type  $X$  which is implemented by the following self-explanatory script:

```
fst :: (X, Y) -> X
fst (x, y) = x
```

In the above example, the program `fst` which is of type  $(X, Y) \rightarrow X$  expects an input of product type  $(X, Y)$ . Functional programming languages such as HASKELL makes use of *pattern matching* to check that the input presented to the program `fst` must possess certain defining characteristic of those data belonging to the product type. Now data belonging to the product type  $(X, Y)$  are of the specific form  $(x, y)$  that comprises two components  $x$  and  $y$ . The program `fst` pattern matches any given input to the form  $(x, y)$  and outputs the first component. We shall see more instance of pattern matching when dealing with list types.

It is important to realize that the function type constructor  $\rightarrow$  is versatile in creating higher-order functions, as the example below shows.

**Example 4 (Higher-order functions)** The addition operation `addint` that takes in two summands of integers to produce their sum can be implemented using the following code:

```
addint :: Int -> Int -> Int
addint x y = x + y
```

Here the higher-order function type  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ , under the convention of right associativity, is the same as  $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ . Indeed given an input of  $x :: \text{Int}$  the function `addint` returns as output a program, i.e., `addint x`, of function type  $\text{Int} \rightarrow \text{Int}$ . This is because `addint x` is waiting to accept an integer argument,  $y :: \text{Int}$ , and returns as output  $x+y$  which is of type  $\text{Int}$ .

Equipped with the above understanding of the syntax `addint x y` and the typing assignment involved, the programmer bears in mind that the function `addint x` is applied to the datum  $y$  instead of the function `addint` is applied to a pair of data  $(x, y)$ .

To understand the higher-order function type better, here is another illustration.

**Example 5 (Higher-order functions)** Consider the evaluation map which takes in as first argument a function  $f :: \text{Int} \rightarrow \text{Int}$ , followed by a second argument an integer  $n :: \text{Int}$ , and evaluates the function  $f$  on  $n$ . Then we can implement the evaluation map `eval` as follows:

```
eval :: (Int -> Int) -> Int -> Int
eval f n = f n
```

In the preceding example, we see HASKELL's facility for creating higher-order functions allows us to pass functions as arguments of other functions. The following example further emphasizes on this point.

**Example 6 (Higher-order functions)** Consider the composition operator which takes in as first argument a function  $f :: A \rightarrow B$ , followed by a second argument another function  $g :: B \rightarrow C$ , and then compose them to give the composite function  $f ; g$ . Then we may implement the composition operator `comp` as follows:

```
comp :: (A -> B) -> (B -> C) -> (A -> C)
comp f g x = g (f x)
```

Indeed `comp` is the HASKELL analogue of the composition operator  $\circ$  in mathematics, namely:

$$\circ : (A \rightarrow B) \rightarrow ((B \rightarrow C) \rightarrow (A \rightarrow C)), (\circ(f))(g) = g \circ f.$$

The practice of passing functions around as arguments of other functions is central and characteristic of functional programming, and because of this, functions have such a special status in functional programming that functional programmers often go by the slogan: “*treat functions as first-class citizens*” [1, p. 51].

### 2.3.3 List type

So far we have encountered several type constructors which are construed as HASKELL analogues of set-theoretic constructors that we are familiar in mathematics. The next type constructor in HASKELL is perhaps more familiar to a computer scientist, rather than a mathematician; it is the list type. Given a type  $A$  we have its *list type*,  $[A]$ . An instance of a datum of type  $[A]$  is a list, i.e., a sequence of elements each of type  $A$ . More precisely, an instance of  $[A]$  can *either* be the empty list  $[\ ]$  that contains nothing *or* a list,  $(a : as)$ , which has its head (the first term) the datum  $a :: A$ , followed by its tail, i.e., another list  $as :: A$  (read as *a*'s).

Because of the innate recursive nature of list types, a program  $p$  that manipulates lists typically reads a finite initial segment of the input list, gives some instructions to output a finite initial segment (typically, the head) and then pass the tail of the input list to be processed by  $p$ ; whence the term *tail recursion* that describes this kind of programming technique.

**Example 7 (List type)** The program `int1` which takes in two lists, e.g.,  $[x_0, x_1, x_2, \dots]$  and  $[y_0, y_1, y_2, \dots]$ , and interleaves their elements to produce  $[x_0, y_0, x_1, y_1, x_2, y_2, \dots]$  can be defined recursively as follows:

```
int1 :: [A] -> [A] -> [A]
int1 [ ] l = l
int1 l [ ] = l
int1 (x:xs) (y:ys) = x:y: int1 xs ys
```

Simply put, the program `int1` takes the heads of the two input lists and interleaves them, and then repeats the procedure on the tails of the two input lists.

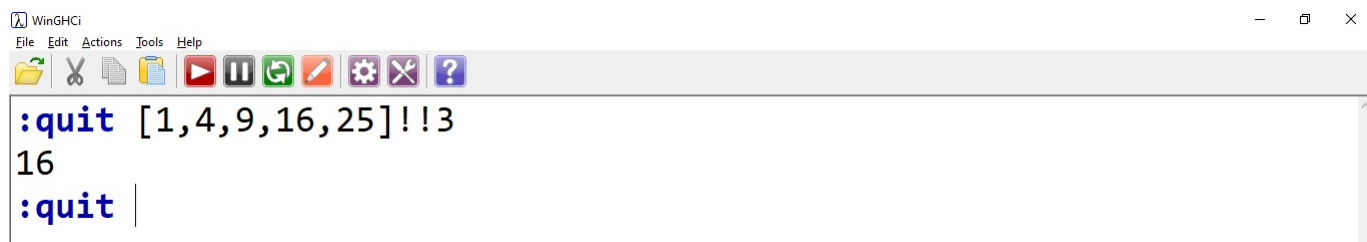
Indeed tail recursion in HASKELL is just another instance of pattern matching mentioned earlier (i.e., right after Example 3). The program `int1` expects a list followed by another. Thus it is either the first list is empty or the second or both are not empty. In third possibility, the input data are



matched against the structure  $(x:xs)$  and  $(y:ys)$  so as to produce the output  $x:y: \text{int1 } xs$   
 $ys$ .

HASKELL has a peculiarity of calling the first element  $x$  of a list  $(x:xs)$  its zeroth element – something a beginner must bear in mind. So, to extract the  $(n + 1)$ th element of a list  $a$ , we use the command `!!` by writing `a !! n`.

**Example 8** *Suppose we want to extract the 4th element of the list  $[1, 4, 9, 16, 25]$ , we can do this:*



```
WinGHCI
File Edit Actions Tools Help
:quit [1,4,9,16,25]!!3
16
:quit |
```

Figure 2: Extracting the 4th element of the list  $[1, 4, 9, 16, 25]$

How does the list type help us understand mathematics better? A simple and straightforward answer is that it enables us to code mathematical sequences (e.g., sequences of real numbers, sequences of closed intervals, sequences of functions, etc.) as data. For instance the data type `[Double]` contains all real number sequences. We shall see how lists coded in this manner can be used to deal with concepts like convergence (c.f. Sections 3.1 and 3.2).

### 2.3.4 Type synonym

Apart from product, function and list constructors, HASKELL has the facility to create *type synonyms*. By this, we mean that we can give a new name for a constructed type.

**Example 9 (Type synonym)** *We may name the constructed product type of a pair of `Int` types as `Pairint` by declaring that:*

```
type Pairint = (Int,Int)
```

*Subsequently, one may then use `Pairint` as a synonym of `(Int, Int)` within the same environment of this type declaration. For instance, we can define the first projection map as follows:*

```
fst :: Pairint -> Int
fst (x,y) = x
```

## 3 Implementing numerical methods

It may appear that the computational power of the small fragment of HASKELL (which we described in the preceding section) is very limited since it merely consists of three ground types (`Int`, `Bool`,

Double) and three type constructors  $((-, -), - \rightarrow -, [-])$ . We now demonstrate that despite its structural leanness this fragment of the language is surprisingly expressive. In particular, we show that it is possible to perform most of the real-number computations that one encounters in an introductory course on elementary numerical methods. The reader is encouraged to compare our functional programs that implement these numerical methods with their imperative counterparts. It is hoped that through this comparison, readers can decide for themselves the merits of functional programming as an alternative programming paradigm to imperative programming. Readers who need to refresh their memory concerning numerical methods may refer to [2] – this book also contains imperative MATLAB codes for implementing all the numerical methods mentioned herein. Since we do not want to deal with exact real arithmetic (i.e., real number computation with arbitrary precision) in this paper, we conveniently use `Double` for real number data type (double-precision) to implement our calculations of real numbers. Readers interested in exact real arithmetic may refer to [5].

Closed intervals play an important role in calculus and real analysis. Coding a closed bounded interval  $[a, b]$  as a pair of double-precision floating point numbers, we call up the type synonym `Interval` to denote the *Cartesian product* of `Double` with itself:

```
type Interval = (Double, Double)
```

In calculus and real analysis, the main subject of study are different classes of functions of the form  $f : \mathbb{R} \rightarrow \mathbb{R}$ , e.g., continuous functions, differentiable functions, and so on. Cognizant that programs of function type are first-class citizens in functional programming, it makes sense for us to create a data type to handle real-valued functions of a single real variable. To do so, we set the type synonym `Function` to represent the function space from `Double` to itself, i.e.,

```
type Function = Double -> Double
```

Of course, we can also create data types for real-valued functions of two real variables, i.e., functions of the form  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ . We can create the type:

```
type F2 = (Double, Double) -> Double
```

However, in view of functions being first-class citizen we would prefer the higher-order construction via the type synonym `Function2` defined by:

```
type Function2 = Double -> Double -> Double
```

In general, instead of realizing the function  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ , we always realize its associated higher-order function

$$\hat{f} : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R}), (\hat{f}(x))(y) = f(x, y).$$

In the literature of functional programming, the act of turning a function  $f : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  into its associated higher-order function  $\hat{f} : \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$  is called the *currying*.

Naming new types, such as `Interval`, `Function` and `Function2`, by using type synonyms not only makes coding a lot more readable but also gives meaning to the various kinds of mathematical objects that the programmer is handling. The following example illustrates this point:

**Example 10** Consider a mathematician programmer writing a HASKELL program `ptwsum` that outputs the pointwise sum of two functions (each of type `Function`). Then `ptwsum` must be of type `Function -> Function -> Function`, i.e., it feeds on a function `f` as its first argument, followed by another function `g` as its second argument, before returning the output of the pointwise sum. The correct typing assignment for the program `ptwsum` helps the programmer focus on the job at hand. In this case the scripts for `ptwsum` are then given by:

```
ptwsum :: Function -> Function -> Function
ptwsum f g x = f(x) + g(x)
```

### 3.1 Point approximation

One of the ways to compute a real number,  $\alpha$ , is by producing a sequence of real numbers,  $\{a_n\}_{n=0}^{\infty}$ , that converges to it. We refer to this computational approach as the *point approximation* approach. The important underlying idea here is to code a sequence of (distinct) real numbers as a list of double-precision floating numbers, `[Double]`. This is where HASKELL's facility of list type plays a crucial role in describing the phenomenon of convergence. More precisely, given a list of `Double` data

$$[a_0, a_1, \dots, a_n, \dots]$$

that represents a convergent sequence  $\{a_n\}_{n=0}^{\infty}$ , we output the first element  $a_n$  if the next element is sufficiently close to it in a *relative* sense, i.e.,

$$\left| \frac{a_n}{a_{n+1}} - 1 \right| < \epsilon, \quad (1)$$

where  $\epsilon > 0$  is the required precision. We call the inequality (1) the *relative precision requirement*. The relative precision requirement has the advantage over the absolute precision requirement, i.e.,

$$|a_n - a_{n+1}| < \epsilon,$$

especially when the magnitudes of  $a_n$  and  $a_{n+1}$  are very small to begin with.

The following program `relerr` is employed to display the first element of the list that satisfies the precision requirement `eps` in the sense of (1):

```
relerr :: Double -> [Double] -> Double
relerr eps (a:b:xs)
  | abs(a/b - 1) < eps = a
  | otherwise          = relerr eps (b:xs)
```

There are two useful features of HASKELL employed in the construct of the above program.

1. `relerr` makes use of tail recursion. Given a relative error of `eps`, the program `relerr` checks whether the first two terms `a` and `b` of the input sequence are relatively close within the given relative error of `eps`. If this is the case, the program `relerr` returns the first term `a` of the input sequence. However if this is not the case, the program `relerr` continues to apply the precision check on the tail of the sequence.

2. The program `relerr` uses guarded environment for piecewise definition. The HASKELL guards are flagged out by the indented vertical bars `|` (appearing in lines 3 and 4 of the above program). Note that the syntax `otherwise` covers the complement of the preceding condition(s) (i.e., it implicitly refers to  $\text{abs}(a/b - 1) < \text{eps} \geq a$ ).

Suppose the relative precision requirement (1) is first met for the index  $n$ , the program `relerr` terminates and returns the element  $a_n$ .

**Remark 11** *In situations where the approximations are not of a small magnitude, then it would have been more accurate to use the absolute precision requirement. This can also be easily implemented in HASKELL as the following program:*

```
abserr :: Double -> [Double] -> Double
abserr eps (a:b:xs)
  | abs(a-b) < eps = a
  | otherwise      = abserr eps (b:xs)
```

*To avoid switching between these two precision requirements, we adhere to the relative precision requirement throughout this paper.*

The above point approximation approach (together with the relative precision requirement) will be applied to yield a HASKELL implementation of the following elementary numerical methods:

1. Fixed point iteration method (Section 3.1.1);
2. Newton-Raphson's method (Section 3.1.2);
3. Numerical differentiation (Section 3.1.3);
4. Euler's method (Section 3.1.4);
5. Runge-Kutta's method (Section 3.1.5); and
6. Numerical integration (Section 3.1.6).

### 3.1.1 Fixed point iteration

Let  $f$  be a continuous function defined on some closed bounded interval  $I$  and  $x_0 \in I$  be given. Then applying  $f$  repeatedly yields the following trajectory of iterates:

$$x_0, f(x_0), f^2(x_0), \dots, f^n(x_0), f^{n+1}(x_0), \dots \quad (2)$$

There is an inherent self-similarity in the above sequence in that sense that when we apply the function  $f$  to every term of the sequence we have

$$f(x_0), f^2(x_0), f^3(x_0), \dots, f^{n+1}(x_0), f^{n+2}(x_0), \dots,$$

which in fact is the tail of the original sequence (2). This observation lends itself directly to the technique of tail recursion in HASKELL, and we thus code the trajectory (2) using the following program:

```
iterates :: Double -> Function -> [Double]
iterates a f = a : (iterates (f a) f)
```

**Example 12** Consider the sequence defined recursively by

$$x_0 = 1, \text{ and } x_{n+1} = 1 + \frac{1}{1 + x_n}.$$

In this case, we see that

$$x_{n+1} = f(x_n),$$

where  $f(x) = 1 + \frac{1}{x}$ ,  $x \neq 0$ . We program  $f$  as follows:

```
f1 :: Function
f1 x = 1 + 1/x
```

The following shows the first 10 terms of the sequence  $(x_n)$  by applying `iterates` on the seed 1 and `f1`:

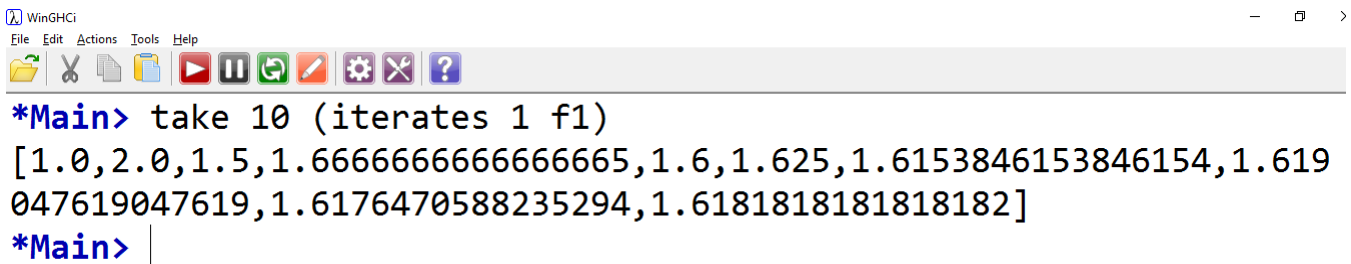


Figure 3: Calculating the first 10 terms of  $(x_n)$

Suppose further that the sequence  $\{f^n(x_0)\}_{n=0}^\infty$  converges, i.e.,  $\lim_{n \rightarrow \infty} f^n(x_0) = \alpha$ . Then by the continuity of  $f$  and the fact that a convergent sequence and any of its sub-sequence share the same limit, we have

$$f(\alpha) = f\left(\lim_{n \rightarrow \infty} f^n(x_0)\right) = \lim_{n \rightarrow \infty} f^{n+1}(x_0) = \lim_{n \rightarrow \infty} f^n(x_0) = \alpha.$$

In other words,  $\alpha$  is a *fixed point* of  $f$ .

**Example 13** A fixed point  $\alpha$  of the function  $f(x) = 1 + \frac{1}{x}$  satisfies the equation

$$1 + \frac{1}{\alpha} = \alpha,$$

which is equivalent to  $\alpha^2 - \alpha - 1 = 0$ . Thus, the possible values of  $\alpha$  are  $\frac{1 \pm \sqrt{5}}{2}$ .

Certain natural conditions are sufficient to guarantee that the trajectory of  $f$ , starting with the seed  $x_0$ , is convergent. One such set of condition is to require  $f$  to be a *contraction mapping*, i.e., there exists a constant  $c \in [0, 1)$  such that

$$\forall x, y \in I. |f(x) - f(y)| \leq c|x - y|.$$

**Example 14** For the case of  $f(x) = 1 + \frac{1}{x}$  defined on the interval  $I = \left[\frac{3}{2}, 2\right]$ , note that

$$f(I) \subseteq I$$

and by the decreasing nature of  $f$  we have

$$f^{n+1}(I) \subseteq f^n(I)$$

for all  $n = 1, 2, \dots$ . Furthermore, by the Mean Value Theorem one can show that

$$\forall x, y \in I. |f(x) - f(y)| \leq \frac{4}{9}|x - y|.$$

Hence  $f$  is a contraction mapping from  $I$  to  $I$ .

Given sufficient conditions that ensure the convergence of the trajectory of  $f$  hold, with seed value  $x_0$ , we can implement the fixed point iteration via the following algorithm:

```
fpi :: Double -> Double -> Function -> Double
fpi eps a f = relerr eps (iterates a f)
```

The program `relerr` ensures that when the consecutive terms  $f^n(x_0)$  and  $f^{n+1}(x_0)$  of the trajectory get relatively close the program `fpi` returns the term  $f^n(x_0)$ .

**Example 15** Returning to Example 12, we employ the program `fpi` to calculate the fixed point  $\alpha = \frac{1 + \sqrt{5}}{2} \in I = \left[\frac{3}{2}, 2\right]$  of  $f(x) = 1 + \frac{1}{x}$  with a relative error of 0.001 (see Figure 4).

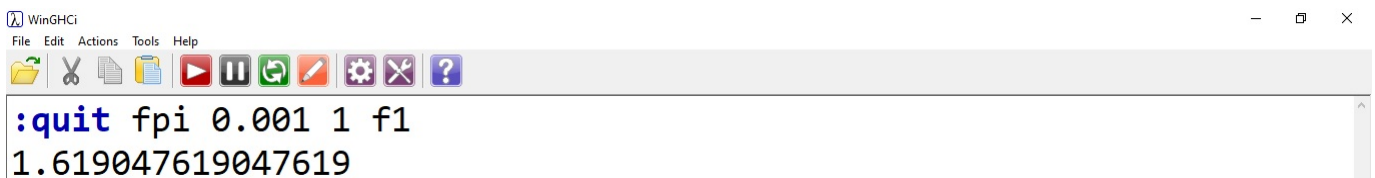


Figure 4: Calculating  $\alpha$  with a relative error of 0.001

### 3.1.2 Newton-Raphson method

A special instance of fixed point iteration is the famous Newton-Raphson iteration scheme for solving numerically the equation

$$f(x) = 0,$$

where  $f$  is a differentiable function. Indeed the iteration scheme, involving the successive constructions of tangents to the curve  $y = f(x)$  and their intersections with the  $x$ -axis, is given by

$$x_{n+1} = g(x_n), \quad n = 0, 1, 2, \dots,$$

where  $g(x) = x - \frac{f(x)}{f'(x)}$ . The upshot here is that each differentiable function  $f$  yields a new function  $g$ . This observation allows us to exploit the functional feature of HASKELL by creating a higher-order function that takes  $f$  as its input and yields  $g$  as its output. This function is realized by the program `newtonit` below:

```
newtonit :: Function -> Function -> Function
newtonit f df x = x - (f x) / (df x)
```

Note that in the above program the derivative of the given function `f` is denoted by `df` (which has to be supplied independently by the user).

As we mentioned in Section 1, pure functions are isolated and do not interfere with other functions. This specific aspect of functional programming allows us to re-use codes freely. In this case, we re-use the `fpi` program that we have written in Section 3.1.1 to implement the Newton-Raphson's method as follows:

```
newton :: Double -> Double -> Function -> Function -> Double
newton eps a f df = fpi eps a (newtonit f df)
```

We have been advertising certain beneficial features of functional programming such as (tail) recursion, passing functions around as arguments, and neater codes by virtue of these features. However, we have not compared the run-time performance of programs written in a functional language versus that in an imperative language.

In what follows, we briefly compare the performance of Newton-Raphson's method implemented in HASKELL (a functional language) and MATLAB (an imperative language). This comparison is done in the context of the following concrete computational problem:

**Example 16** *In his book *Flos*, Leonardo de Pisa, better known as Fibonacci (1170–1250), was able to closely approximate the positive solution to the cubic equation*

$$x^3 + 2x^2 + 10x = 20.$$

*Let us put `newton` to the test by calculating the root  $\alpha$  up to  $10^{-9}$  as Fibonacci did more than 700 years ago.*

Rearranging the original equation we have

$$f(x) = 0,$$

where  $f(x) := x^3 + 2x^2 + 10x - 20$ . The derivative of  $f$  is given by

$$f'(x) = 3x^2 + 4x + 10.$$

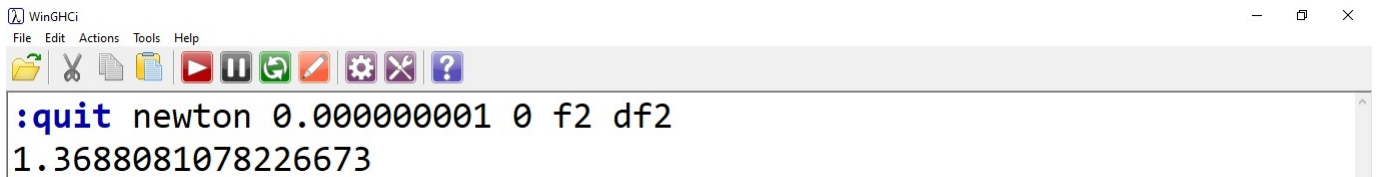
We now create the HASKELL programs for these two functions which we denote by `f2` (for  $f$ ) and `df2` (for  $f'$ ) respectively:

```
f2 :: Function
f2 x = x**3 + 2*(x**2) + 10*x - 20

df2 :: Function
df2 x = 3*(x**2) + 4*x + 10
```

Note that the binary operation `**` stands for exponentiation of the numbers in `Double`.

Running the HASKELL program `newton` with these inputs, with the seed  $x_0 = 0$  and a relative error of `0.000000001`, we have:



```
WinGHCI
File Edit Actions Tools Help
:quit newton 0.000000001 0 f2 df2
1.3688081078226673
```

Figure 5: Calculating  $\alpha$  the root of the Fibonacci's cubic equation using `newton.hs`

The Newton-Raphson's method can be implemented in an imperative language such as MATLAB using the following modified codes taken from [2, p. 52]:

```
clear; clc;
eps = 1.0e-9;
x = 0;
xnext = x - f(x)/(fd(x));
d = abs(xnext-x);
while (d > eps)
    x=xnext;
    xnext = x - f(x)/(fd(x));
    d = abs(x/xnext - 1);
end
fprintf('%18.16f', xnext);
```

Running the above MATLAB program `newton.m` on the following functions `f.m` and `fd.m`



```

%% function f(x)
function y=f(x)
y = x^3 + 2*(x^2) + 10*x - 20

%% function fd(x)
function y=fd(x)
y = 3*(x^2) + 4*x + 10
    
```

yields the following output:

```

Editor - C:\MATLAB\R2017a\bin\newton.m
newton.m  x  f.m  fd.m  +
1 - clear; clc;
2 - eps = 1.0e-9;
3 - x = 0;
4 - xnext = x - f(x)/(fd(x));
5 - d = abs(xnext-x);
6 - while (d > eps)
7 -     x=xnext;
8 -     xnext = x - f(x)/(fd(x));
9 -     d = abs(x/xnext - 1);
10 - end
11 - fprintf('%18.16f \n',xnext);

Command Window
1.3688081078213725
fx >>
    
```

Figure 6: Calculating  $\alpha$  the root of the Fibonacci’s cubic equation using newton.m

Figure 7 compares the run-time of the two programs, and gives evidence that the run-time of the HASKELL implementation (functional: 0.000 seconds) is very much faster than that of the MATLAB implementation (imperative: 0.006 seconds).

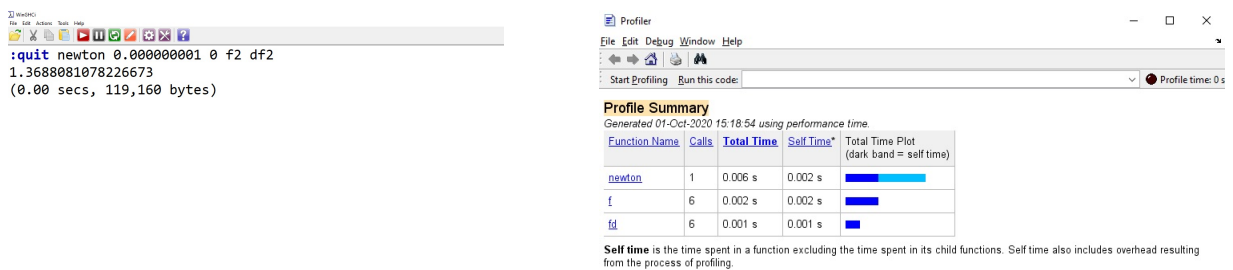


Figure 7: Comparison of run-time of HASKELL (left) and MATLAB programs for computing  $\alpha$

### 3.1.3 Numerical differentiation

We now move on to numerical differentiation. A preliminary estimate of the gradient of the tangent to the curve  $y = f(x)$  at the point  $x$  is given by the first order quotient, i.e., the gradient of the secant

line joining the points  $(x, f(x))$  and  $(x + h, f(x + h))$ , i.e.,  $\frac{f(x+h)-f(x)}{h}$ , where  $h$  is a ‘small enough’ for which  $f$  is defined in the interval  $[x, x + h]$ . This is easily calculated by the following program:

```
secant :: Double -> Function -> Function
secant h f x = (f (x+h) - f x) / h
```

We can simulate the convergence of  $h$  to 0 by successively halving the initial difference of  $h$ . In particular, we make use of the following sequence

$$h, \frac{1}{2} \cdot (h), \frac{1}{2} \cdot \left(\frac{1}{2}h\right), \frac{1}{2} \cdot \left(\frac{1}{2} \cdot \left(\frac{1}{2}h\right)\right), \dots$$

that converges to 0. This sequence is embedded in a recursive manner into the program `secantseq` to calculate the corresponding secant values of each term of the above sequence:

```
secantseq :: Double -> Function -> Double -> [Double]
secantseq h f x = (secant h f x) : secantseq (h/2) f x
```

To calculate the derivative of  $f$  at  $x$ , we set the initial value of  $h$  to be 1 and demand the consecutive secant values to be relatively close via the program below:

```
easydiff :: Double -> Function -> Function
easydiff eps f x = relerr eps (secantseq 1 f x)
```

**Example 17** We now apply the `easydiff` program to find the derivative of  $f(x) = \sin(x)$  at  $x = 0$ .

```
f3 :: Function
f3 x = sin (x)
```

We set the error  $10^{-9}$  and the output is given in Figure 8 below:

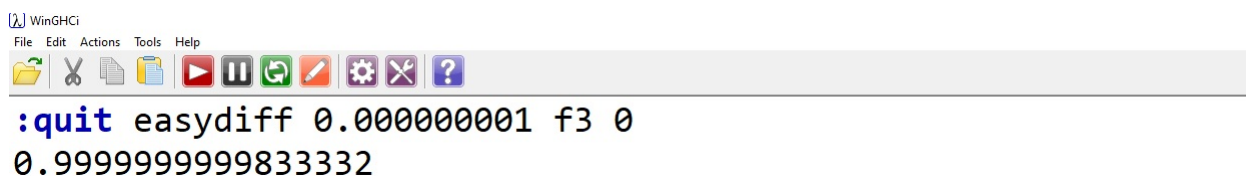


Figure 8: Calculating the derivative of  $\sin(x)$  at  $x = 0$

### 3.1.4 Euler’s method

Given the differential equation

$$\frac{dy}{dx} = f(x, y), \quad y(a) = y_0,$$

the problem is to find an approximation to  $y$  when  $x = b$ .

Let  $n$  be the number of steps,  $x_0 = a$ ,  $x_n = b$ , and the step size

$$h := \frac{x_n - x_0}{n}.$$

Then

$$y_k = y_{k-1} + hf(x_{k-1}, y_{k-1}), \quad k = 0, 1, \dots, n, \quad (3)$$

Using the recurrence relation, it is intended that  $y(b)$  be calculated as  $y_n$ .

Equation 3 implies that for a given pair  $(x_{k-1}, y_{k-1})$ , we can calculate  $y_k$  using the formula

$$y_k = g(x_{k-1}, y_{k-1}),$$

where  $g(x, y) = y + h \cdot f(x, y)$ . This observation translates into the following `ynext` program:

```
ynext :: Double -> Double -> Double -> Function2 -> Double
ynext h x y f = y + (h * (f x y))
```

We may now unwind the recursive formula (3) one step at a time to produce the following sequence  $\{y_0, y_1, y_2, \dots\}$  which can be explicitly expressed as:

$$y_0, y_1 := g(x_0, y_0), y_2 := g(\underbrace{x_0 + h}_{x_1}, y_1), y_3 := g(\underbrace{x_1 + h}_{x_2}, y_2), \dots,$$

The above sequence is generated by the following recursive function:

```
eulerseq :: Double -> Double -> Double -> Function2 -> [Double]
eulerseq h x y f = w : (eulerseq h (x+h) w f)
  where w = (ynext h x y f)
```

We then extract the  $n$ th term,  $y_n$ , of the sequence `euler a b y0 f` using the operation `!!` (see Example 8):

```
euler :: Double -> Double -> Double -> Function2 -> Int -> Double
euler a b y0 f n = (eulerseq h a y0 f) !! n
  where h = (b-a) / fromIntegral n
```

**Example 18** We will proceed to use Euler's method to solve the following differential equation

$$\frac{dy}{dx} = 2x + 2y, \quad y(0) = 1$$

for the value of  $y(1)$  choosing  $n = 10000$ .

```
f4 :: Double -> Double -> Double
f4 x y = 2*x + 2*y
```

Employing the program `euler` on the inputs  $a = 0$ ,  $b = 1$ ,  $y_0 = 1$  and  $n = 10000$ , we obtain: Note that the exact value of  $y(1)$  is  $\frac{3}{2}(e^2 - 1) \approx 9.58338414839597$ .

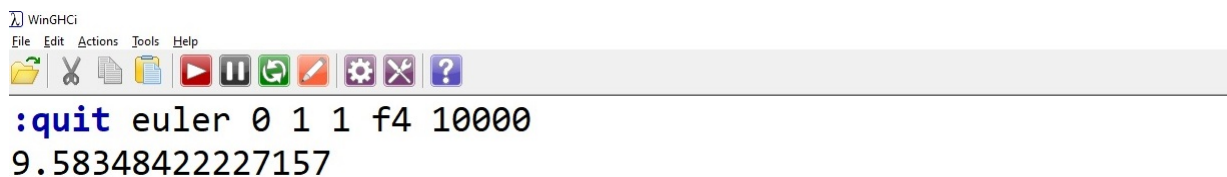


Figure 9: Calculating the value of  $y(1)$  using Euler method

### 3.1.5 Runge-Kutta 4th Order

The most widely known member of the *Runge-Kutta family* is generally referred to as “*RK4*”, the “classic Runge-Kutta method” or simply as “the *Runge-Kutta method*”.

Consider the initial value problem specified below:

$$\frac{dy}{dx} = f(x, y), \quad y(x_0) = y_0.$$

We aim to find a numerical approximation of  $y$  at  $x = b$ .

As before, let  $n$  be the number of steps,  $x_0 = a$ ,  $x_n = b$ , and the step size  $h$  be given by

$$h := \frac{x_n - x_0}{n}.$$

Then define

$$\begin{aligned} x_{k+1} &= x_k + h, \\ y_{k+1} &= y_k + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4), \end{aligned}$$

for  $k = 0, 1, 2, 3, \dots, n$ , using

$$\begin{aligned} k_1 &= f(x_k, y_k), \\ k_2 &= f\left(x_k + \frac{h}{2}, y_k + h \cdot \frac{k_1}{2}\right), \\ k_3 &= f\left(x_k + \frac{h}{2}, y_k + h \cdot \frac{k_2}{2}\right), \\ k_4 &= f(x_k + h, y_k + hk_3). \end{aligned}$$

From its definition, we see that the RK4 method is similar in structure to that of the Euler’s method.

Let us proceed to code the  $k_j$ ’s ( $j = 1, 2, 3, 4$ ) first.

```

k1 :: Double -> Double -> Function2 -> Double
k1 x y f = f x y

k2 :: Double -> Double -> Double -> Function2 -> Double
k2 h x y f = f (0.5*h+x) (y+0.5*(k1 x y f)*h)

k3 :: Double -> Double -> Double -> Function2 -> Double
k3 h x y f = f (0.5*h+x) (y+0.5*(k2 h x y f)*h)

k4 :: Double -> Double -> Double -> Function2 -> Double
k4 h x y f = f (x+h) (y+(k3 h x y f) *h)

```

Then we code the weighted-sum of these  $k_j$ 's, i.e.,

$$k_1 + 2k_2 + 2k_3 + k_4$$

using the following program:

```

ksum :: Double -> Double -> Double -> Function2 -> Double
ksum h x y f = (k1 x y f) + 2*(k2 h x y f)
               + 2*(k3 h x y f) + (k4 h x y f)

```

Finally, we mimic the recursive style of `ynext` to create the `rk4` program below:

```

rk4 :: Double -> Double -> Double -> Function2 -> Double
rk4 h x y f = y + (1/6)*h*(ksum h x y f)

```

This is then used to generate the list of  $y_k$ 's:

```

rk4seq :: Double -> Double -> Double -> Function2 -> [Double]
rk4seq h x y f = w: rk4seq h (x+h) w f
  where w = (rk4 h x y f)

```

Finally, we extract  $y_n$  as the  $n$ th term of the above list:

```

rk4meth :: Double -> Double -> Double -> Function2
         -> Int -> Double
rk4meth a y0 b f n = (rk4seq h a y0 f)!!n
  where h = (b-a)/fromIntegral(n)

```

**Example 19** We now use `rk4meth` to solve the following differential equation:

$$\frac{dy}{dx} = f(x, y),$$

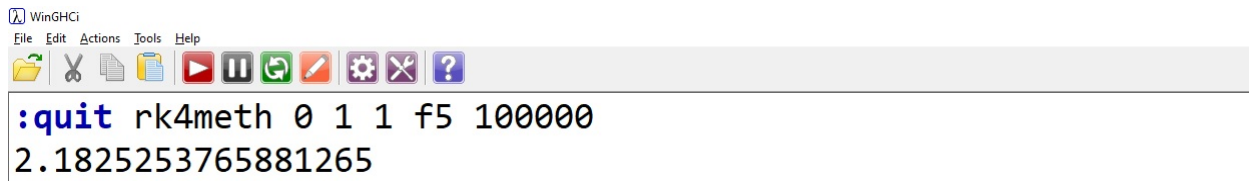


Figure 10: Calculating the value of  $y(1)$  using RK4 method

where  $f(x, y) = 3e^{-x} - 0.4y$  with initial condition  $x_0 = 0$  and  $y_0 = 1$ .

Note that the exact value of  $y(1)$  is  $-5e^{-1} + 6e^{-0.4} \approx 2.18252307035662$ .

### 3.1.6 Numerical integration

The (trapezoidal) area under the line segment joining  $(a, f(a))$  and  $(b, f(b))$  provides a first approximation of the definite integral of the continuous function  $f$  over the interval  $[a, b]$ . This common numerical method is called the *simple trapezoidal rule*. For instance, if  $f(x) = x^2$  for  $x \in [1, 2]$ , then the trapezoidal area under the line segment  $y = 3x - 2$  over  $[1, 2]$  estimates the value of the definite integral  $\int_1^2 x^2 dx$  (see Figure 11).

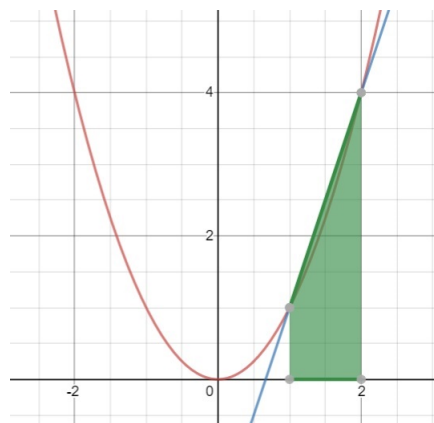


Figure 11: Simple trapezoidal rule for estimating  $\int_1^2 x^2 dx$

This first estimate,  $\frac{1}{2}(b - a)(f(a) + f(b))$ , can be calculated using `trap`:

```

trap :: Function -> Interval -> Double
trap f (a,b) = (f(a)+f(b)) * (b-a) / 2
    
```

The idea for calculating the definite integral is to see it as the limit of the sum of trapezia obtained by a certain refinement procedure which we illustrate below. From the first estimation of  $\int_a^b f(x) dx$  produced by the simple trapezoidal rule on  $[a, b]$ , we may then produce the second estimation of the definite integral by applying the following procedure:

1. Bisect the interval  $I := [a, b]$  into two equal sub-intervals  $I_0 := [a, c]$  and  $I_1 := [c, b]$ , where  $c = \frac{a+b}{2}$ .
2. Apply the simple trapezoidal rule to estimate:
  - (i)  $\int_a^c f(x) dx$  over the interval  $I_0$ ; and
  - (ii)  $\int_c^b f(x) dx$  over the interval  $I_1$ .
3. Add the two estimates in (2) to obtain the second estimation of  $\int_a^b f(x) dx$ .

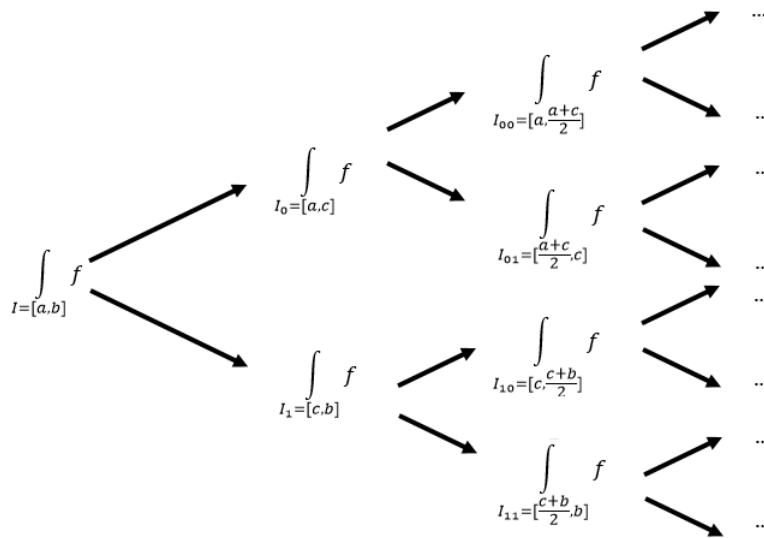


Figure 12: Binary tree of definite integrals yielded from the refinement

We can think of the first application of Step (2) as a branching at Level 1 that yields leaves, i.e.,  $\int_a^c f(x) dx$  arising from the left half interval  $I_0$  and  $\int_c^b f(x) dx$  arising from the right half interval  $I_1$  respectively. To yield the third estimate, we apply the above procedure, i.e., Steps (1)–(3), to estimate each of the aforementioned two definite integrals. This sequential refinement yields the binary tree below (Figure 12), where the trapezoidal rule is applied at each node to estimate the definite integral located there.

More precisely, there are  $2^2$  trapezium each estimating the definite integrals over the  $2^2$  definite integrals at Level 2. In general, the  $n$ th estimate is given by the total area of all the  $2^n$  trapezia arising from iteratively applying Step (2) at Level  $n$ .

We now turn to the task of finding the total area  $A_n$  of all the  $2^n$  trapezia at Level  $n$ , whence forming the sequence  $\{A_n\}_{n=0}^\infty$ . The idea here is to imagine a zip is located at each node, and as the zip moves forward it adds the elements of the two emergent branches up. For this purpose, we devise a program called `zipadd`.

```
zipadd :: [Double] -> [Double] -> [Double]
zipadd (x0:xs) (y0:ys) = (x0 + y0) : zipadd xs ys
```

The iterative procedure we have illustrated above lends itself to HASKELL's capability to handle tail recursion. Thus, the sequence of estimates  $\{A_n\}_{n=0}^{\infty}$  can be represented by the list produced by the following program:

```
integralseq :: Function -> Interval -> [Double]
integralseq f (a,b) = (trap f (a,b)):
                    zipadd (integralseq f (a,c))
                        (integralseq f (c,b))
                    where c = (a+b)/2
```

Lastly, we rely on the usual technique of picking up the first term of the above sequence which is relatively close to the next one, up to the given relative precision requirement.

```
integrate :: Double -> Function -> Interval -> Double
integrate eps f (a,b) = relerr eps (integralseq f (a,b))
```

**Example 20** Let us use the program `integrate` to calculate  $\int_1^2 x^2 dx$  with a relative error of  $10^{-7}$ .

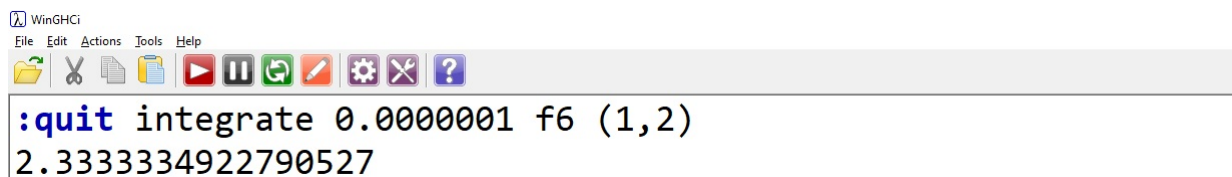


Figure 13: Calculating the value of  $\int_1^2 x^2 dx$  using the trapezoidal rule

The exact value of  $\int_1^2 x^2 dx$  is  $\frac{7}{3} \approx 2.333333333333333$ .

The enthusiastic reader is invited to program using HASKELL the Simpson's quadrature. Such a program can simply replace the trapezoidal program `trap` in the program `integralseq`, thus illustrating the re-usability of codes in HASKELL.

### 3.2 Interval approximation

Another approach of computing a real number,  $\alpha$ , is by computing an decreasing family,  $\mathcal{C}$ , of closed intervals  $\{I_n\}_{n=0}^{\infty}$ , i.e.,

$$I_0 \supseteq I_1 \supseteq I_2 \supseteq I_3 \supseteq \dots I_n \supseteq I_{n+1} \supseteq \dots$$

such that the lengths of the intervals converge to 0 and  $\alpha \in \bigcap_{n=0}^{\infty} I_n$ .

The key idea here is to set a precision requirement on the calculation of  $\alpha$  based on the length of the approximating interval. Given a descending family of closed intervals  $\{(a_n, b_n)\}_{n=0}^{\infty}$  coded as

$$[(a_0, b_0), (a_1, b_1), (a_2, b_2), \dots]$$



select the first interval  $(a_n, b_n)$  such that the endpoints are close to each other in the usual relative sense, i.e.,

$$\left| \frac{a_n}{b_n} - 1 \right| < \epsilon,$$

where the positive quantity,  $\epsilon$ , is the user-set precision requirement. This can be achieved by the program below:

```
relerrint :: Double -> [Interval] -> Double
relerrint eps ((a,b):xs)
  | abs(a/b - 1) < eps = a
  | otherwise          = relerrint eps xs
```

**Remark 21** As noted in Remark 11, one may employ absolute precision requirement if the two end points are not of small magnitudes. In that case, the following program is suited for this purpose.

```
abserr :: Double -> [Double] -> Double
abserr eps (a:b:xs)
  | abs(a-b) < eps = a
  | otherwise      = abserr eps (b:xs)
```

We deal with numerical solution of equations here. For convenience, we assume that  $f$  is a continuous function defined on  $[a, b]$  such that  $f$  has exactly one real zero,  $\alpha$ , in  $[a, b]$ .

**Definition 22 ( $f$ -good interval)** An interval  $I = [a, b]$  is  $f$ -good if  $f$  experiences a change of sign over it, i.e.,  $f(a)f(b) \leq 0$ .

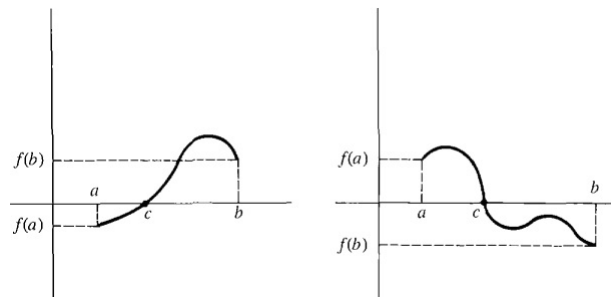


Figure 14:  $[a, b]$  is  $f$ -good

We can implement  $f$ -goodness of an interval as follows:

```
good :: Function -> Interval -> Bool
good f (a,b) = ((f a)*(f b) <= 0)
```

### 3.2.1 Bisection method

Consider any descending chain,  $\mathcal{C}$ , of  $f$ -good intervals ordered by reverse inclusion:

$$[a_0, b_0] \supseteq [a_1, b_1] \supseteq [a_2, b_2] \supseteq \dots [a_k, b_k] \supseteq [a_{k+1}, b_{k+1}] \supseteq \dots,$$

where  $\lim_{k \rightarrow \infty} [a_k, b_k] = 0$ . By the Heine-Borel theorem,  $\bigcap_{i=0}^{\infty} [a_i, b_i] = \{\alpha\}$ . In other words,  $\{\alpha\}$  is the supremum of the descending chain  $(\mathcal{C}, \supseteq)$ .

For each given  $f$ -good interval  $[a, b]$ , its left (respectively, right) half sub-interval is  $[a, c]$  (respectively,  $[c, b]$ ), where  $c = \frac{a+b}{2}$ . The bisection method selects the  $f$ -good subinterval of these two sub-intervals; in the event when both sub-intervals are  $f$ -good, then the left one is always selected.

```
goodhalf :: Function -> Interval -> Interval
goodhalf f (a,b) = let c = (a+b)/2 in
                    if (good f (a,c)) then (a,c) else (c,b)
```

Based on the bisection method, we output the desired descending chain of  $f$ -good subintervals in the form of a list of intervals, beginning with the input  $f$ -good interval  $[a_0, b_0]$  using the algorithm below:

```
bisectseq :: Function -> Interval -> [Interval]
bisectseq f (a,b) = (a,b) : bisectseq f (goodhalf f (a,b))
```

Lastly, we set an error of tolerance  $\text{eps}$  such that as soon as the relative ratio,  $a/b$  of the endpoints of an approximating interval  $[a, b]$  differs from 1 by this set error then the right endpoint of this interval is displayed.

```
bisect :: Double -> Function -> Interval -> Double
bisect eps f (a,b) = relerrint eps (bisectseq f (a,b))
```

**Example 23** We revisit Example 16, i.e., solve the Fibonacci's cubic equation using the bisection method on the interval  $[1, 2]$ . We set the accuracy to a relative error of  $10^{-9}$ .

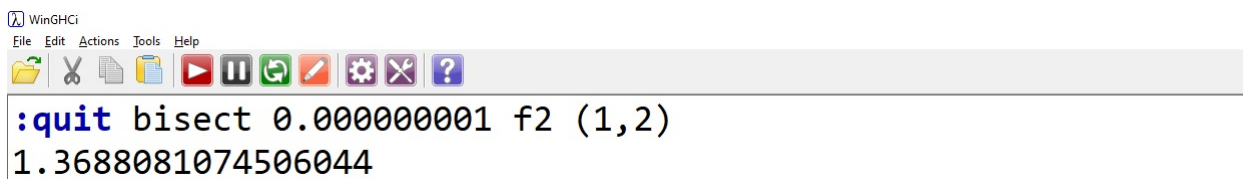


Figure 15: Applying the bisection method to solve Fibonacci's cubic equation

### 3.2.2 Linear interpolation

For each given  $f$ -good interval  $[a, b]$ , the line which interpolates between the points  $(a, f(a))$  and  $(b, f(b))$  cuts the  $x$ -axis to form a better estimate,  $c$ , for  $\alpha$ , the zero of  $f$ , where  $c = \frac{af(b) - bf(a)}{f(b) - f(a)}$ .

The HASKELL version of the saying the above is:

```
linecut :: Function -> Interval -> Double
linecut f (a,b) = (a*f(b)-b*f(a))/(f(b)-f(a))
```

This ‘cut-point’  $c$  creates the left ‘half’ subinterval  $(a, c)$  and right ‘half’ subinterval  $(c, b)$ . Like in the bisection method, we can select the  $f$ -good ‘half’ subinterval of these two:

```
linehalf :: Function -> Interval -> Interval
linehalf f (a,b) = let c = linecut f (a,b) in
                    if (good f (a,c)) then (a,c) else (c,b)
```

Again, we create a descending family of  $f$ -good intervals relying on the linear interpolation method.

```
linehalfseq :: Function -> Interval -> [Interval]
linehalfseq f (a,b) = (a,b) : linehalfseq f (linehalf f (a,b))
```

Setting the precision requirement yields the desired algorithm:

```
linint :: Double -> Function -> Interval -> Double
linint eps f (a,b) = relerrint eps (linehalfseq f (a,b))
```

**Example 24** We employ the above program `linint` to (again) solve the Fibonacci’s cubic equation using the linear interpolation method on the interval  $[1, 2]$  with a relative error of  $10^{-9}$ .

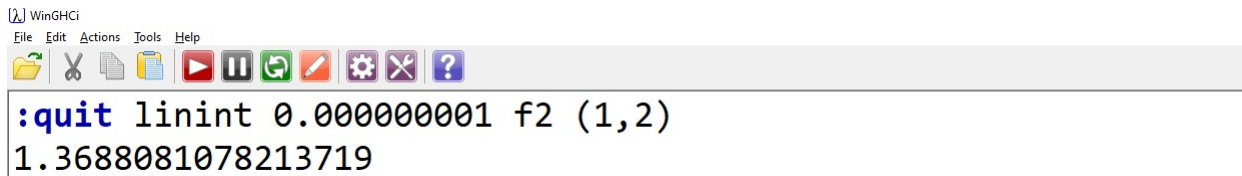


Figure 16: Applying the linear interpolation method to solve Fibonacci’s cubic equation

## 4 Conclusion

### 4.1 Summarizing what we have so far

This paper is a beginner’s tutorial to functional programming via HASKELL, and results from a rewriting of an earlier conference-paper version contributed by the second author back in ATCM 2017 ([6]).

In this paper we advertise several attractive features of the functional programming paradigm (using Haskell as a vehicular language) with the aim of enticing mathematicians in this community

to adopt functional programming paradigm as a new way of thinking about these numerical solutions of old problems. A mathematician HASKELL programmer enjoys the following benefits:

1. **Mathematical objects as data.** HASKELL can handle familiar mathematical objects such as natural numbers, real numbers,  $n$ -tuples, sequences, intervals and functions as data. Indeed a native environment within HASKELL has been set up in Section 2.3 to think about sets as data types. Mathematician programmers can now calculate with functions, sequences and intervals by treating them directly as data, i.e., executing transformations on them as HASKELL expressions of the respective data types.
2. **Programs as functions.** HASKELL programs can be thought (and handled) as functions which anticipate inputs and manufacture outputs. The functional programming paradigm is thus a natural computing paradigm for mathematicians. The functional aspect of HASKELL, together with the typing discipline enforced by this language, helps mathematician programmers focus on the salient data types involved at the input and the output respectively.
3. **Purity and composability.** Complex programs can be built by composing simple pure functions in a sequential manner – just as one would compose two mathematical functions one after another to produce a more complicated one. All functions are isolated and do not influence one another. Thus, the mathematician programmer decomposes a complex problem into simpler isolated subtasks to be realized as pure functions, and to piece them together by functional composition. Additionally, purity of functions encourages the re-usability of codes without worry that re-used functions will affect the performance of current ones (see Section 3.1.2).
4. **Higher-order functions.** HASKELL’s ability to perceive and handle functions as data allows mathematician programmers to pass functions around as inputs of higher-order functions (see Section 2.3.2). This gives mathematician programmers a convenient platform to handle familiar higher-order operators, such as the differential operator, the integral operator, the summation operator, etc.
5. **Pattern matching.** HASKELL employs pattern matching for the programmer to check an input data for the characteristic form or structure pertaining to the data type it is suppose to belong. This allows the mathematician programmer to assign the output according to the form of the input data (see Examples 3 and 7).
6. **Guarded environment.** Mathematician programmers can now script piecewise defined functions, i.e., those of the form

$$f(x) = \begin{cases} \dots & \text{if } \dots \\ \dots & \text{if } \dots \end{cases},$$

directly using HASKELL guards (see Section 3.1).

7. **Recursion.** Throughout the paper, we have exploited recursion when dealing with recursively defined structures; for instance, we write programs using tail recursion for list types (see Section 2.3.3). Mathematician programmers find this a welcomed feature because recursion, as well as induction, is a central theme of mathematics.

In Section 3, all the above features have been employed in our HASKELL implementation of several elementary numerical methods. Compared to their counterparts, programs written in functional style read directly as mathematical functions – hence more suited to the taste of a mathematician programmer. In addition, run-time efficiency is not compromised in functional programming (see Section 3.1.2 for a comparison with an imperative language).

## 4.2 Looking into the future

The next step for this research is to demonstrate HASKELL’s capability of handling Complex Analysis. The authors are now working out the algorithms for calculating contour integrals of complex-valued functions, such as

$$\int_{C[z_0;R]} f(z) dz,$$

where  $C[z_0; R]$  denotes the circle of center  $z_0 \in \mathbb{C}$  and radius  $R > 0$ , and  $f(z)$  is a complex-valued function of a single complex variable  $z$ . In particular, we hope to employ the famous Cauchy Integral Formula (and its extension) to create algorithms for numerical differentiation.

Another important calculation in Complex Analysis that needs to be implemented using HASKELL is that of Taylor (respectively, Laurent) series. The reason for this choice is that series representation of complex-valued functions is a cornerstone of Complex Analysis, enabling one to understand the property of holomorphicity.

## References

- [1] Abelson, H., & Sussman, G. J. Formulating Abstractions with Higher-Order Procedures. In *Structure and Interpretation of Computer Programs* (p. 51), MIT Press, 1984.
- [2] Ang, K. C. *Numerical Mathematics with MATLAB*, Prentice-Hall, 2009.
- [3] Bird, B. *Introduction to Functional Programming using Haskell*, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.
- [4] Bird, B. *Thinking Functionally with Haskell*, Cambridge University Press, October 2014.
- [5] Ho, W. K. Exact Real Calculator for Everyone. In W.-C. Yang, M. Majweski, T. Alwis, and I. K. Rana, (Eds.) *Proceedings of the 18th Asian Technology Conference in Mathematics* (pp. 1-15). Bombay, India: ATCM, December 2013.
- [6] Ho, W. K. Appreciating functional programming: A beginner’s tutorial to haskell illustrated with applications in numerical methods. In W.-C. Yang, D. B., Meade, & Y. Yuan, (Eds.) *Proceedings of the 22nd Asian Technology Conference in Mathematics* (pp. 50-64). Taiwan, Zhong Li: ATCM, December 2017.
- [7] Hutton, G. *Programming in Haskell*, 2nd Edition, Cambridge University Press, 2016.

- [8] Microsoft documentation. *Functional programming vs. imperative programming (LINQ to XML)*. Retrieved from <https://docs.microsoft.com/en-us/dotnet/standard/linq/functional-vs-imperative-programming>.

## A Getting started with HASKELL

As a person who hates ploughing through instruction manuals, I find the following steps manageable for most Microsoft Windows users who wants to dive into HASKELL without much fuss. (Other OS users can follow the instructions at <https://www.haskell.org/platform/>.)

1. You need a text editor. While a simple Notepad would do, Notepad++ has many welcomed features (e.g., color-coding).
2. You need a HASKELL compiler. For this paper, we use GHC (Glasgow Haskell Compiler). The best way is to download the Haskell Platform. Visit <https://www.haskell.org/platform/>. You need to follow the instructions published there *to the letter*.
3. GHC takes a Haskell script, i.e., a program that has a `.hs` extension, and compile it.
4. GHC has an interactive mode (called GHCi) which allows you to interactively interact with scripts. To do this, type `ghci` at your prompt. Alternatively, you can click on `ghci.exe` if you go through Windows. You will land up with this:

```
GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
Prelude>
```

5. In your text editor, write your HASKELL script such as the following:

```
myfact :: Int -> Int
myfact 0 = 1
myfact n = n * myfact (n-1)
```

Save it as `baby.hs` or something, and in the same folder which `ghci` was invoked.

6. Once inside GHCi, load `baby.hs` by typing `:l baby`. Then you can play with `myfact`; see below:

```
ghci> :l baby
[1 of 1] Compiling Main          ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> myfact 4
24
```

7. If you like a Windows GUI for GHCi, you can visit <https://github.com/haskell/winghci/wiki/Installation> for the manual installation instructions. This is optional.