
Title	A Haskell implementation of the Lyness-Moler's numerical differentiation algorithm
Author(s)	Weng Kin Ho and Chu Wei Lim
Source	<i>Electronic Proceedings of the 25th Asian Technology Conference in Mathematics, 14 – 16 December 2020, Radford, Virginia, USA, and Thailand, 83-98</i>

Copyright © 2020 Mathematics & Technology, LLC

A HASKELL Implementation of the Lyness-Moler's Numerical Differentiation Algorithm

Weng Kin Ho and Chu Wei Lim

wengkin.ho@nie.edu.sg, chuwei.lim@aostudies.com.sg

National Institute of Education
Nanyang Technological University
Singapore

Abstract

This paper describes a computational problem encountered in numerical differentiation. By restricting the problem to a proper subclass of differentiable functions, a numerical solution first proposed by Lyness and Moler is considered and implemented in the functional programming language HASKELL. The accuracy of the calculation of the numerical derivative using the Lyness-Moler's method crucially lies in our recursive algorithm for computing contour integrals.

1 Introduction

To obtain the derivative of a (differentiable) function f at a point $x = a$ numerically one can rely on the definition of the derivative by first principles, i.e.,

$$f'(a) := \lim_{h \rightarrow 0} \frac{f(a+h) - f(a)}{h},$$

whenever the limit exists, and calculates the quotient $\frac{f(a+h) - f(a)}{h}$ by h gets arbitrarily close to 0 ([3]). However, the computation of the derivative using this numerical method is plagued by a problem, which we illustrate using Example 1.

Example 1 *Let us consider our familiar exponential function*

$$f(x) = \exp(x), \quad x \in \mathbb{R}.$$

To model the convergence of h to 0, we set $h_n = 10^{-n}$ ($n = 1, 2, 3, \dots$), and the gradient s_n of the secant line through the points $(a, \exp(a))$ and $(a + h_n, \exp(a + h_n))$ is given by

$$s_n = \frac{\exp(a + h_n) - \exp(a)}{h_n}.$$

In this case, we calculate $\exp'(a)$ as the following limit:

$$\lim_{n \rightarrow \infty} s_n = \lim_{n \rightarrow \infty} \frac{\exp(a + h_n) - \exp(a)}{h_n}.$$

An EXCEL spreadsheet can be designed to record the following quantities:

(i) n , (ii) $h_n = 10^{-n}$, (iii) $\exp(a + h_n) - \exp(a)$, and (iv) s_n .

A sample of such an EXCEL spreadsheet design is shown in Figure 1, with $a = 1$.

	A	B	C	D
1	a	1		
2	n	$\exp(a+1/10^n)-\exp(a)$	$1/10^n$	$s(n)$
3	0	4.67077427	1	4.670774
4	1	0.285884195	0.1	2.858842
5	2	0.027319187	0.01	2.731919
6	3	0.002719641	0.001	2.719641
7	4	0.000271842	0.0001	2.718418
8	5	2.7183E-05	0.00001	2.718295
9	6	2.71828E-06	0.000001	2.718283
10	7	2.71828E-07	1E-07	2.718282
11	8	2.71828E-08	1E-08	2.718282
12	9	2.71828E-09	1E-09	2.718282
13	10	2.71828E-10	1E-10	2.718283
14	11	2.71831E-11	1E-11	2.718314
15	12	2.71871E-12	1E-12	2.718714
16	13	2.71783E-13	1E-13	2.717826
17	14	2.70894E-14	1E-14	2.708944
18	15	0	1E-15	0
19	16	0	1E-16	0
20	17	0	1E-17	0
21	18	0	1E-18	0
22	19	0	1E-19	0
23	20	0	1E-20	0

Figure 1: EXCEL spreadsheet showing the calculation of s_n 's

Because the exponential function is computed using floating-point arithmetic, both the difference $\exp(a + h_n) - \exp(a)$ or the quantity h_n will eventually become 0 as n tends to infinity, i.e., when h_n gets sufficiently small (e.g., in our program, after sufficiently many ‘halvings’ of h). In this instance, the quantity $\exp(a + h_n) - \exp(a)$ becomes 0 when $n = 15$, and thus so does s_n . As a result, the computed values of $\{s_n\}$ do not converge to $\exp'(1) = \exp(1) \approx 2.71828 \dots$. This computational problem is quite disturbing as it rears its ugly head even for very simple functions such as $\exp(x)$.

The aforementioned computational problem is not new, and there are several methods proposed to overcome it. In this paper, we want to zoom into a specific method first introduced by Lyness and Moler in 1967 ([8]), and later developed and improved upon by others such as [5]. This method makes clever use of the famous Cauchy Integral Formula which every mathematics major must learn in the tertiary mathematics education (probably in some Complex Analysis module).

The purpose of this paper is to give a brief explanation of the Lyness-Moler’s method and to implement this method in the form of an algorithm written in HASKELL (a functional language). By so doing, this paper improves on a certain part of the work done earlier in [6] with regard to numerical differentiation. Furthermore, the recursive style of writing programs in HASKELL is exploited in the evaluation of a certain contour integral. We thus enjoy a higher accuracy in numerical calculations, together with an economy of codes.

There are some pre-requisites needed to understand this paper. Firstly, we assume that the reader knows HASKELL or any functional programming language. For a quick introduction to

HASKELL that would help in the understanding of this paper, we point the interested reader to our recent introductory tutorial [7]. The benefits of using HASKELL in connection to mathematics (e.g., higher accuracy and economy of codes) are also advertised in that paper; we have no space to discuss those here. Secondly, the reader requires some knowledge in Complex Analysis – at least up to the Cauchy Integral Formula, which can be looked up in any standard Complex Analysis textbook such as [4]. For the convenience of our target audience, we present in Section 2 a quick summary of the relevant results that already appeared in [7], and also the theory developed in [8]. In Section 4, we implement the Lyness-Moler’s method in HASKELL as promised. Throughout the paper, we use GHC (Glasgow Haskell Compiler) and the Windows GUI, WinGHCi, for HASKELL. Readers who wish to try out the HASKELL programs presented here will find the quick guide on “Getting started with HASKELL” in the Appendix of [7] useful.

2 Preliminaries

In [7], the numerical method for calculating the derivative described in Section 1 was implemented in HASKELL. We explain very briefly the functional programming paradigm and syntax of HASKELL as we go through the codes for the program `easydiff` below.

In HASKELL the real numbers (represented in double-precision) are of the `Double` data type. The first step is to calculate the gradient of the secant line that passes through the points $(x, f(x))$ and $(x + h, f(x + h))$, where h is intended to be a real number to be made as small as we wish.

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

Notice that `secant` is assigned the higher-order type `Double -> Function -> Double -> Double` as it is a function that feeds on a real number h , followed by the function f and lastly the real number x and then returns the value of $\frac{f(x+h) - f(x)}{h}$.

To simulate the limiting process of h tending to 0, we exploit the list data type `[Double]` of real numbers with the intention of producing terms of a sequence that approximates the desired limit. In this case, we want to compute the sequence defined by

$$s_n := \frac{f(x + 10^{-n}) - f(x)}{10^{-n}}.$$

We may thus define the so-called *secant-sequence* $\{s_n\}_{n=1}^{\infty}$ recursively:

```
secantseq :: Double -> Function -> Double -> [Double]
secantseq h f x = (secant h f x) : secantseq (h/10) f x
```

For a convergent sequence $\{a_n\}_{n=1}^{\infty}$, we choose the absolute error precision requirement given by:

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

i.e., for a given error of $\epsilon > 0$, one continues to calculate the next term a_{n+1} so long as the above inequality holds. The absolute error can be calculated using the following program:

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

Making use of all these preceding programs, the program `easydiff` given below calculates an approximation of $f'(x)$ at the relative precision of $\epsilon > 0$:

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

Because the data type `Double` relies on the floating point representation, it follows that `secantseq` suffers from the same problem illustrated in Example 1 (see Figure 2). More precisely, the 20th iterate of the gradient of the secant line, s_{20} , equals 0.



Figure 2: Same problem encountered in `secantseq`

We make crucial (and essential) use of the integration program `integrate` introduced in [7], which we reproduce here. This program is constructed based on trapezoidal rule.

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

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

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

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

Let us look at the following example of evaluating a certain definite integral.

Example 2 *Evaluate the definite integral*

$$\int_0^1 e^{\cos(2\pi t)} \cos(\sin(2\pi t)) \, dt.$$

To evaluate this integral, we write a short functional program `fun1` to code the integrand:

```
fun1 :: Double -> Double
fun1 x = (exp(cos(2*pi*x)))*(cos(sin(2*pi*x)))
```

Running `integrate` on `fun1` over $[0, 1]$ with an absolute error of 10^{-10} yields:

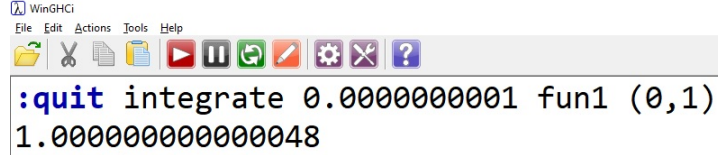


Figure 3: Evaluating $\int_0^1 e^{\cos(2\pi t)} \cos(\sin(2\pi t)) dt$

Using techniques in complex analysis, it can be shown that

$$\int_0^1 e^{\cos(2\pi t)} \cos(\sin(2\pi t)) dt = 1.$$

How does the program `integrate` work? The idea here is to calculating the definite integral as the limit of the sum of trapezia obtained by successive refinements 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 procedure:

1. Bisect the interval $I_0 := [a, b]$ into two equal sub-intervals $I_{10} := [a, c]$ and $I_{11} := [c, b]$, 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_{10} ; and
 - (ii) $\int_c^b f(x) dx$ over the interval I_{11} .

3. Add the two estimates in (2) to obtain the second estimation of $\int_a^b f(x) dx$.

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 and $\int_c^b f(x) dx$ arising from the right half interval 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. More precisely, there are 2^2 trapezium each estimating the definite integrals over the 2^2 definite integrals at Level 2 (see Figure 4). 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 .

In order to total up the area of these 2^n trapezia, the following componentwise-addition of two lists of real numbers comes in handy:

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

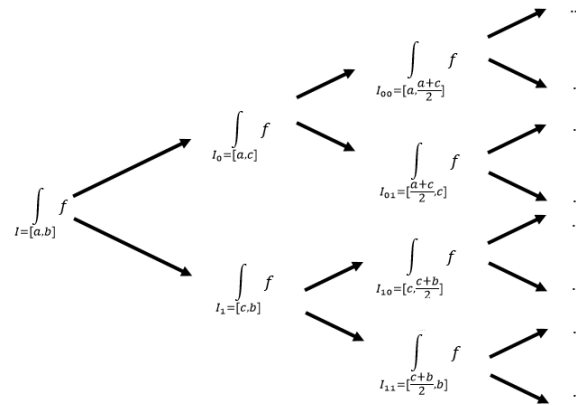


Figure 4: Branching process in calculating the estimate for the definite integral

The iterative procedure we have illustrated yields a sequence of approximations (making use of more and more trapezia), and this sequence can be 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 precision requirement.

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

However, there are some integrand functions for which the above precise requirement can cause a premature termination with a erroneous output.

Example 3 Evaluate the definite integral $\int_0^1 \cos^2(2\pi t) dt$.

The exact value of this definite integral is $\frac{1}{2}$. Running the program `integralseq` on $g(t) = \cos^2(2\pi t)$ over the interval $[0, 1]$ and taking the first 5 terms, the returns are shown in Figure 5.

Figure 5: Sequence of approximations of the definite integral

Observe that the first two terms of the above sequence of approximations to the area under the graph $y = g(x)$ are equal to 1.0. This happens because of the shape of the graph of the

function g (see Figure 6), i.e., the end-point values of the function at the boundaries of the first partition and the second one are equal.



Figure 6: Shape of the graph $y = g(x)$ for $0 \leq x \leq 1$

As a result, when one runs the program `integrate` on the above sequence the iteration stops prematurely causing a substantial error (despite setting $\epsilon = 0.0001$) in the estimation of the definite integral (see Figure 7).

```
WinGHCi
File Edit Actions Tools Help
:quit integrate 0.0001 (\t -> (cos(2*pi*t))^2) (0,1)
1.0
```

Figure 7: Wrong calculation of $\int_0^1 \cos^2(2\pi t) dt$

Thus, in practice, the first 5 terms of the sequence of approximations are dropped off to reduce the risk of an premature program termination due to a constant initial segment of certain sequences. The modified code is:

```
integrate :: Double -> Function -> Interval -> Double
integrate eps f (a,b) = abserr eps (drop 5 (integralseq f (a,b)))
```

Figure 8 shows the result of running the modified `integrate` program.

```
WinGHCi
File Edit Actions Tools Help
:quit integrate 0.0001 (\t -> (cos(2*pi*t))^2) (0,1)
0.5
```

Figure 8: Correct calculation of $\int_0^1 \cos^2(2\pi t) dt$

For lack of space here, we point our readers to the full exposition of `integrate`, together with all the mathematics that supports it, in Section 3.1.6 of [7]. Essential use of `integrate` is made in Section 4.2.

3 Lyness-Moler's method

The idea behind Lyness-Moler's method¹ of numerical differentiation is quite nifty. One transforms the task of finding the derivative to that of finding a certain definite integral. Numerical integration is then applied to evaluate this definite integral. Since numerical integration involves limiting sums, *not* quotients, the problem described in Section 1 is thus circumvented. To effect this transformation, one crucially relies on the famous Cauchy Integral Formula (CIF, for short) which every mathematics major must have learnt in some Complex Analysis course.

Theorem 4 (Cauchy Integral Formula) *Let $f : \mathbb{C} \rightarrow \mathbb{C}$ be a complex-valued function of one complex variable. Suppose that f is holomorphic on and inside a simple, closed and positively oriented contour C . Then f is infinitely differentiable. Furthermore, for any $a \in \mathbb{C}$ that lies in the interior of the region bounded inside C , it holds that*

$$f^{(n)}(a) = \frac{n!}{2\pi i} \oint_C \frac{f(z)}{(z-a)^{n+1}} dz.$$

Notice that there is a price to pay. We have not completely solved the original problem because the Cauchy Integral formula requires f to be holomorphic. So for the Lyness-Moler's method to work, one must extend the real-valued function $f(x)$ of a single real variable to a *holomorphic* complex-valued function $f(z)$ of a single complex variable. This in fact is a *huge* restriction on the class of functions (the derivative of which we are calculating numerically): there are functions f that are infinitely differentiable on \mathbb{R} which are not even continuous on \mathbb{C} (see Example 5)!

Example 5 *Consider the function*

$$g(x) = \begin{cases} e^{-1/x} & \text{if } x > 0; \\ 0 & \text{if } x \leq 0. \end{cases}$$

Note that $g^{(n)}(0) = 0$ for all $n = 0, 1, 2, \dots$. Thus,

$$\sum_{n=0}^{\infty} \frac{g^{(n)}(0)}{n!} x^n = 0$$

for all $x \in \mathbb{R}$. Since $g(x) \neq 0$ on $x > 0$, it follows that g is not even real-analytic. Notice also that the complex-valued function of a single complex variable z defined by

$$f(z) = \begin{cases} e^{1/z} & \text{if } z \neq 0; \\ 0 & \text{otherwise,} \end{cases}$$

is not even continuous at $z = 0$, let alone holomorphic.

¹The author independently thought of this method about a decade ago, but only to discover it existed way back in the 1960s while writing up this paper.

Although the Lyness-Moler's method can only be used on a smaller class of differentiable functions, it suffices for most of our considerations. Besides, our main objective is to implement this numerical differentiation in `HASKELL` and so we are not too bothered by this restriction.

Let $f(x)$ be a real-valued function of a single real variable x and assume that $f(z)$ is holomorphic on and inside the circle C . By the Cauchy Integral Formula, for any $a \in \mathbb{C}$ that lies in the interior of the region bounded inside C it holds that

$$a_n := \frac{f^{(n)}(a)}{n!} = \frac{1}{2\pi i} \oint_C \frac{f(z)}{(z-a)^{n+1}} dz, \quad n = 0, 1, 2, \dots,$$

In particular, for the circle $C = C[a; r] := \{z \in \mathbb{C} : |z-a| = r\}$ we can parametrize C as follows:

$$z - a = re^{2\pi ti}, \quad 0 \leq t \leq 1.$$

Direct substitution yields:

$$\begin{aligned} a_n &= \frac{1}{2\pi i} \oint_{C[a;r]} \frac{f(z)}{(z-a)^{n+1}} dz \\ &= \frac{1}{2\pi i} \int_0^1 \frac{f(re^{2\pi ti} + a)}{r^{n+1}e^{2(n+1)\pi ti}} \cdot 2\pi i r e^{2\pi ti} dt \\ &= \frac{1}{r^n} \int_0^1 f(re^{2\pi ti} + a) e^{-2n\pi ti} dt, \quad n = 0, 1, 2, \dots, \end{aligned}$$

This gives us the formula for a_n :

$$a_n = \frac{1}{r^n} \int_0^1 f(re^{2\pi ti} + a) e^{-2n\pi ti} dt, \quad n = 1, 2, 3, \dots \quad (1)$$

Since $f(z)$ is holomorphic on and inside the circle $C[a; r]$, so $g(z) := (z-a)^{2n} f(z)$. Because $g^{(n)}(a) = 0$, it follows that

$$0 = \frac{1}{2\pi i} \oint_{C[a;r]} \frac{(z-a)^{2n} f(z)}{z^{n+1}} dz.$$

This simplifies to

$$0 = r^n \int_0^1 f(re^{2\pi it} + a) e^{2n\pi ti} dt.$$

Since $r \neq 0$, we would have

$$\frac{1}{r^n} \int_0^1 f(re^{2\pi ti} + a) e^{2n\pi ti} dt = 0, \quad n = 1, 2, 3, \dots \quad (2)$$

By Euler's theorem, $e^{2n\pi ti} = \cos(2n\pi t) + i \sin(2n\pi t)$, and so adding Equations 1 and 2 we have:

$$a_n = \frac{2}{r^n} \int_0^1 f(re^{2\pi ti} + a) \cos(2n\pi t) dt, \quad n = 1, 2, 3, \dots \quad (3)$$

All in all, we have arrived at the following theorem:

Theorem 6 *Let $f(x)$ be a real-valued function of one real variable x . Suppose that $f(z)$ is holomorphic on and inside the circle $C[a; r] := \{z \in \mathbb{C} : |z - a| = r\}$. Then*

$$f(a) = \int_0^1 f(re^{2\pi ti} + a) dt,$$

and

$$f^{(n)}(a) = \frac{2(n!)}{r^n} \int_0^1 f(re^{2\pi ti} + a) \cos(2n\pi t) dt, \quad n = 1, 2, 3, \dots$$

Furthermore, when $a \in \mathbb{R}$, then

$$f(a) = \int_0^1 g(t) dt, \quad (4)$$

and

$$f^{(n)}(a) = \frac{2(n!)}{r^n} \int_0^1 g(t) \cos(2n\pi t) dt, \quad n = 1, 2, 3, \dots, \quad (5)$$

where $g(t) = \text{Re}(f(re^{2\pi ti} + a))$.

In the next section, we channel our energy to implement the n th derivative calculator in HASKELL by evaluating numerically those definite integrals in Equation (5) of Theorem 6.

4 HASKELL implementation

We organize the implementation of the Lyness-Moler's method in HASKELL into two subsections. In Subsection , we prepare all the necessary algorithms for elementary calculations in \mathbb{C} , and in Subsection , we develop the Lyness-Moler's numerical differentiation algorithm.

4.1 Elementary calculations in \mathbb{C}

Instead of loading the standard module `Complex` in HASKELL, we choose to build the data type for complex numbers from scratch because building these data types in HASKELL using more primitive means yield better insight and understanding of complex numbers.

For the rectangular representation of complex numbers z in the form $x + iy$ or equivalently the formal pair (x, y) , we use the following type synonym:

```
type Complexr = (Double, Double)
```

For the polar form of complex numbers $z = re^{i\theta}$ or equivalently the formal pair (r, θ) , we use the type declaration:

```
type Complexp = (Double, Double)
```

In this paper, we require only to convert the polar form (r, θ) to the rectangular form (x, y) using the standard equations:

$$x = r \cos \theta \quad (6)$$

$$y = r \sin \theta \quad (7)$$

The conversion program `fromp` is given below:

```
fromp :: Complexr -> Complexp
fromp (r,t) = (r*cos(t),r*sin(t))
```

The real and imaginary parts are realized as follows:

```
re :: Complexr -> Double
re (x,y) = x

im :: Complexr -> Double
im (x,y) = y
```

Setting up the basic arithmetic in the rectangular form is routine.

```
addr :: Complexr -> Complexr -> Complexr
addr (x1,y1) (x2,y2) = (x1+x2,y1+y2)

minusr :: Complexr -> Complexr -> Complexr
minusr (x1,y1) (x2,y2) = (x1-x2,y1-y2)

multr :: Complexr -> Complexr -> Complexr
multr (x1,y1) (x2,y2) = (x1*x2-y1*y2,x1*y2+y1*x2)

divr :: Complexr -> Complexr -> Complexr
divr (x1,y1) (x2,y2) = ((x1*x2+y1*y2)/(x2**2+y2**2),
                        (y1*x2-x1*y2)/(x2**2+y2**2))

intpower :: Complexr -> Int -> Complexr
intpower z 0 = (0,1)
intpower z n = multr z (intpower z (n-1))
```

The four arithmetic operations of addition, subtraction, multiplication and division are implemented as `addr`, `minusr`, `multr` and `divr`. Raising a complex number to a non-negative integral power is realized by `intpower`.

Taking complex conjugate can also be easily implemented as follows:

```
conj :: Complexr -> Complexr
conj (x,y) = (x,-y)
```

Polynomial functions are straightforward to code. Given a list of complex coefficients a_0, a_1, \dots, a_n , the polynomial

$$P(z) := a_0 + a_1z + \dots + a_nz^n$$

can be coded recursively as `polynom`:

```
polynom :: [Complexr] -> (Complexr -> Complexr)
polynom [] z = (0,0)
polynom (x:xs) z = addr x (multr z (polynom xs z))
```

The famous Möbius transformation

$$M_{a,b,c,d}(z) = \frac{az + b}{cz + d},$$

where a, b, c and $d \in \mathbb{C}$ are such that $ad - bc \neq 0$, can also be implemented in **HASKELL**:

```
mobius :: Complexr -> Complexr -> Complexr -> Complexr
        -> (Complexr -> Complexr)
mobius a b c d z = divr (addr (multr a z) b) (addr (multr c z) d)
```

Three complex transcendental functions, namely,

$$\begin{aligned}\cos(z) &= \cos x \cosh y - i \sin x \sinh y, \\ \sin(z) &= \sin x \cosh y + i \cos x \sinh y, \\ \exp(z) &= e^x (\cos y + i \sin y)\end{aligned}$$

need to be in place as well:

```
cosr :: Complexr -> Complexr
cosr (x,y) = ((cos x)*(cosh y), -(sin x)*(sinh y))

sinr :: Complexr -> Complexr
sinr (x,y) = ((sin x)*(cosh y), (cos x)*(sinh y))

expr :: Complexr -> Complexr
expr (x,y) = ((exp x)*cos(y), (exp x)*(sin y))
```

4.2 **HASKELL** implementation of Lyness-Moler's method

Let $f(x)$ be a real-valued function of one real variable x , and suppose that $f(z)$ is holomorphic on and inside the circle $C[a; r] := \{z \in \mathbb{C} : |z - a| = r\}$. According to Theorem 6, for any real number a , it holds that

$$f^{(n)}(a) = \frac{2(n!)}{r^n} \int_0^1 g(t) \cos(2n\pi t) dt, \quad n = 1, 2, 3, \dots, \quad (8)$$

where $g(t) = \operatorname{Re}(f(re^{2\pi ti} + a))$.

The factorial function is easily implemented in **HASKELL**

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)
```

We need to build the integrand $g(t) \cos(2n\pi t)$ first. Here is how do it:

```
build :: (Complexr -> Complexr) -> Integer -> Double -> Double
      -> (Double -> Double)
build f n a r = \t -> ((cos (2*pi*fromIntegral(n)*t)) *
                        re(f(a+r*cos(2*pi*t),r*sin(2*pi*t))))
```

The reader may wish to note that `fromIntegral` is an in-built HASKELL function that converts the integer `n` to a real number of type `Double` so that the multiplication operation `*` is legitimate.

Then we realize Equation (8) as follows:

```
nderiv :: (Complexr -> Complexr) -> Integer -> Double
      -> Double -> Double -> Double
nderiv f n a r eps = k * (integrate eps (build f n a r) (0,1))
      where k = (2*fromIntegral(fact n))/(r^n)
```

5 Sample runs

In this section, we test-run our program `nderiv` on three real-valued functions of one real variable (that satisfy the assumptions of Theorem 6):

1. $f_1(x) = e^x$;
2. $f_2(x) = \cos(x)$;
3. $f_3(x) = \sin(x)$; and
4. $f_4(x) = \frac{e^x}{\cos^3(x) + \sin^3(x)}$.

Example 7 We revisit Example 1 that describes the computational issue arising from the numerical differentiation of $f_1(x) = e^x$ using the usual quotient of differences.

The function $f_1(x) = e^x$ can be extended to the complex exponential function

$$\exp(z) = \exp(x + iy) := e^x(\cos(x) + i \sin(x)),$$

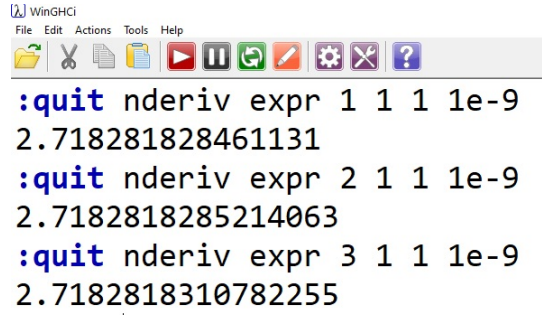
which is entire in \mathbb{C} . For convenience, we choose the circle $C[a, 1]$ with radius 1, centred at the real number $a = 1$. We apply the program `nderiv` to compute the first, second and third derivatives of $f_1(x) = e^x$ at $x = a = 1$ at the precision of $\epsilon = 1 \times 10^{-9}$.

Next we evaluate the first, second and third derivatives of the cosine and sine functions.

Example 8 The function $f_2(x) = \cos(x)$ can be extended to the complex cosine function

$$\cos(z) = \cos(x + iy) := \cos x \cosh y - i \sin x \sinh y,$$

which is entire in \mathbb{C} . For convenience, we choose the circle $C[a, 1]$ with radius 1, centred at the real number $a = 0$. We apply the program `nderiv` to compute the first, second and third derivatives of $f_2(x) = \cos(x)$ at $x = a = 0$ at the precision of $\epsilon = 1 \times 10^{-9}$.

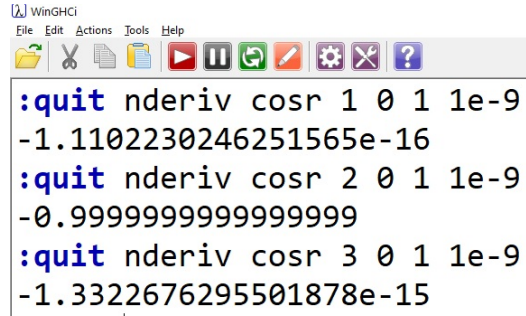


```

WinGHCI
File Edit Actions Tools Help
:quit nderiv expr 1 1 1 1e-9
2.718281828461131
:quit nderiv expr 2 1 1 1e-9
2.7182818285214063
:quit nderiv expr 3 1 1 1e-9
2.7182818310782255

```

Figure 9: Calculating the 1st, 2nd and 3rd derivatives of $f_1(x) = e^x$ at $x = 1$



```

WinGHCI
File Edit Actions Tools Help
:quit nderiv cosr 1 0 1 1e-9
-1.1102230246251565e-16
:quit nderiv cosr 2 0 1 1e-9
-0.9999999999999999
:quit nderiv cosr 3 0 1 1e-9
-1.3322676295501878e-15

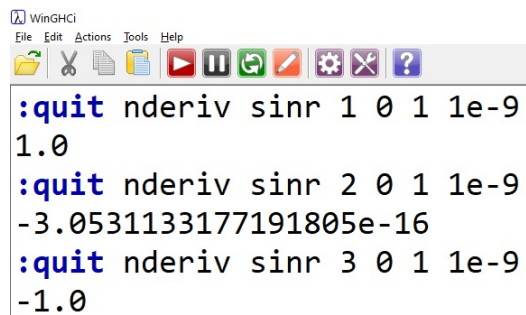
```

Figure 10: Calculating the 1st, 2nd and 3rd derivatives of $f_2(x) = \cos(x)$ at $x = 0$

Example 9 The function $f_3(x) = \sin(x)$ can be extended to the complex sine function

$$\sin(z) = \sin(x + iy) := \sin x \cosh y + i \cos x \sinh y,$$

which is entire in \mathbb{C} . For convenience, we choose the circle $C[a, 1]$ with radius 1, centred at the real number $a = 0$. We apply the program `nderiv` to compute the first, second and third derivatives of $f_3(x) = \sin(x)$ at $x = a = 0$ at the precision of $\epsilon = 1 \times 10^{-9}$.



```

WinGHCI
File Edit Actions Tools Help
:quit nderiv sinr 1 0 1 1e-9
1.0
:quit nderiv sinr 2 0 1 1e-9
-3.0531133177191805e-16
:quit nderiv sinr 3 0 1 1e-9
-1.0

```

Figure 11: Calculating the 1st, 2nd and 3rd derivatives of $f_3(x) = \sin(x)$ at $x = 0$

Example 10 In the 1967 paper by Lyness and Moler who was the first to introduce their numerical differentiation method the authors chose the test function

$$f_4(x) = \frac{e^x}{\cos^3(x) + \sin^3(x)},$$

which they justified to be “tedious to differentiate analytically, but easy to compute for complex arguments ([8, p. 207]). Their calculation on [8, p. 209] yields

$$f^{(5)}(0) = -168.5 \dots,$$

whereas the exact value is an integer, i.e.,

$$f^{(5)}(0) = -164.$$

Since $f_4(x)$ can be extended to the complex function

$$f_4(z) = \frac{e^z}{\cos^3(z) + \sin^3(z)},$$

which is holomorphic on and inside the circle $C[a, 0.5]$ with radius 0.5 (to avoid the singularity), centred at the real number $a = 0$.

We code the function f_4 in the form of the program `testfn`.

```
testfn :: Complexr -> Complexr
testfn z = divr (expr z) (addr (intpower (sinr z) 3) (intpower (cosr z) 3))
```

We apply the program `nderiv` to compute the fifth derivative of $f_4(x) = \frac{e^x}{\cos^3(x) + \sin^3(x)}$ at $x = a = 0$ at the precision of $\epsilon = 1 \times 10^{-9}$ (Figure 12).



Figure 12: Calculating $f_4^{(5)}(0)$

Notably, our HASKELL implementation of the Lyness-Moler’s method produced a more accurate result than the original one presented on [8, p. 208] for their chosen test function.

6 Conclusion

In this paper, we drew wisdom from a 1967 method introduced by Lyness and Moler used to produce accurate numerical calculations of higher-order derivatives of a certain class of differentiable functions, i.e., those which can be extended to holomorphic functions on certain circular regions in the complex plane. The upshot here is that the Lyness-Moler method can be implemented in the functional language HASKELL, and because of the recursive style of writing programs we enjoy higher accuracy and efficiency with an economy of codes.

Lyness-Moler’s method assumes that the integrand can be extended analytically, and so the original problem described in Section 1 is solved for a very restricted class of functions. A paper by Fornberg ([5]) actually removed this assumption, i.e., one need not make use of any additional properties of the integrand function. Our next step would be to study and implement the Fornberg’s method of numerical differentiation in HASKELL.

References

- [1] Bird, B. *Introduction to Functional Programming using Haskell*, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.
- [2] Bird, B. *Thinking Functionally with Haskell*, Cambridge University Press, October 2014.
- [3] Burden, R. L., and Faires, J. D. (2000). *Numerical Analysis*, (7th Ed), Brooks/Cole. ISBN 0-534-38216-9.
- [4] Brown, J. W., and Churchill, R. V. (2014). *Complex Variables and Applications*, (9th Ed), McGraw-Hill Education. ISBN 978-0-07-338317-0.
- [5] Fornberg, B. (1981). Numerical Differentiation of Analytic Functions, *ACM Transactions on Mathematical Software (TOMS)*.
- [6] Ho, W. K. (2017). Appreciating functional programming: A beginner's tutorial to HASKELL illustrated with applications in numerical methods. In Yang, W.-C. Meade, D. B., & Yuan, Y. (Eds.), *Proceedings of the Twenty-second Asian Technology Conference in Mathematics*, 1, 50–64.
- [7] Lim, C. W., and Ho, W. K. (2020). Appreciating functional programming: A beginner's tutorial to HASKELL illustrated with applications in numerical methods. *The Electronic Journal of Mathematics and Technology* 14(1). ISSN 1933-2823.
- [8] Lyness, J. N., and Moler, C. B. (1967). Numerical differentiation of analytic functions. *SIAM J. Numer. Anal.* 4: 202–210. doi:10.1137/0704019.