Learn Physics with Functional Programming

—

Scott N. Walck

Graham Strickland

March 15, 2025

4 Describing Motion

4.2 For $f(x) = x^3$, we have $f'(x) = 3x^2$, so that the relative error is defined by

$$\begin{aligned}
\text{err}(x,a) &= \left| \frac{\frac{f(x+a/2)-f(x-a/2)}{a} - f'(x)}{f'(x)} \right| \\
&= \left| \frac{\frac{(x+a/2)^3-(x-a/2)^3}{a} - 3x^2}{3x^2} \right| \\
&= \left| \frac{\begin{array}{l}[x^3 + (3x^2a)/2 + (3xa^2)/4 + a^3/8] \\ \quad - [x^3 - (3x^2a)/2 + (3xa^2)/4 - a^3/8]\end{array}}{\frac{a}{3x^2}} - 3x^2 \right| \\
&= \left| \frac{\frac{3x^2a + a^3/4 - 3x^2a}{a}}{3x^2} \right| \\
&= \left| \frac{\frac{a^2}{4}}{3x^2} \right| \\
&= \left| \frac{a^2}{12x^2} \right| \\
&= \frac{a^2}{12x^2},
\end{aligned}$$

since $a^2 \geq 0$ and $x^2 \geq 0$.

Thus we have an error of 1 percent if

$$\begin{aligned}
&\mathrm{err}(x, a) = 0.01 \\
\Leftrightarrow\ &\frac{a^2}{12x^2} = 0.01 \\
\Leftrightarrow\ &a^2 = 0.12x^2 \\
\Leftrightarrow\ &a = |x|\sqrt{0.12}
\end{aligned}$$

Then, for $x = 4$, we have

$$\begin{aligned}
a &= 4\sqrt{0.12} \\
&\approx 1.3856406460551018
\end{aligned}$$

and for $x = 0.1$, we have

$$\begin{aligned}
a &= 0.1\sqrt{0.12} \\
&\approx 3.4641016151377546 \times 10^{-2}.
\end{aligned}$$

4.3 Suppose we have a function f and independent variable, say x, such that `derivative 0.01 f x` produces at least a 10 percent error, $\mathrm{err}(x, \epsilon)$, compared to the exact derivative, $f'(x)$. Then, we have

$$\begin{aligned}
\mathrm{err}(x, \epsilon) &= \mathrm{err}(x, 0.01) \\
&= \left| \frac{\frac{f(x+\epsilon/2)-f(x-\epsilon/2)}{\epsilon} - f'(x)}{f'(x)} \right| \\
&= \left| \frac{\frac{f(x+0.01/2)-f(x-0.01/2)}{0.01} - f'(x)}{f'(x)} \right| \\
&= \left| \frac{\frac{f(x+0.005)-f(x-0.005)}{0.01} - f'(x)}{f'(x)} \right| \\
&\geq 0.1.
\end{aligned}$$

If we substitute $\epsilon = 0.01$, $x = \pi \approx 3.141592653589793$, and $f(x) = \cos x$ into the above, we have $f'(x) = -\sin x$, so that

$$\begin{aligned}
\mathrm{err}(x, \epsilon) &= \mathrm{err}(\pi, 0.01) \\
&= \left| \frac{\frac{\cos(\pi+0.005)-\cos(\pi-0.005)}{0.01} + \sin\pi}{-\sin\pi} \right| \\
&\approx 1.0 \\
&\geq 0.1.
\end{aligned}$$

4.4 We cannot apply our error function in 4.3 to `derivative a cos`, since it results in division by 0, but at values close to `t = 0`, we see an initial increase in the error as we move from small values of `a` which stop increasing as we increase `a` by multiples of 10 past `a = 10`.

For the following definition of the error function

```
err :: (R -> R) -> (R -> R) -> R -> R -> R
err f df t a = abs ((derivative a f t - df t) / df t)
```

we have the following output from `ghci`:

```
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 0.01
4.166661374023121e-6
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 0.1
4.166145864253412e-4
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 1
4.114892279159413e-2
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 10
1.191784854932627
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 100
1.0052474970740786
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 1000
1.000935543610645
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 10000
1.000197593287754
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 100000
1.0000199968037815
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 1000000
0.999999644337597
ghci> err (\x -> cos x) (\x -> -sin x) 0.1 10000000
1.000000195308493
```

5 Working with Lists

5.4 We have the function `range` with the following definition in `src/ch05/range.hs`:

```
range :: Int -> [Int]
range x = if x >= 0
          then [0..x]
          else [x..0]
```

`range` returns a list containing all the integers between the argument (inclusive) and 0 in increasing order, i.e, range$(2) = 0, 1, 2$, range$(-4) = -4, -3, \ldots, 0$, and range$(0) = 0$.

We demonstrate as follows:

```
ghci> range (-4)
[-4,-3,-2,-1,0]
ghci> range 2
[0,1,2]
```

```
ghci> range (-4)
[-4,-3,-2,-1,0]
ghci> range 0
[0]
```

We have the function `null'` with the following definition in `src/ch05/null.hs`::

5.5
```
import Data.Foldable

null' :: (Foldable t) => t a -> Bool
null' xs = case toList xs of
  [] -> True
  (_ : _) -> False
```

`null'` returns `True` if an argument `t` of type `a` is empty, otherwise `False`. Since we are using the `Foldable` type, we import `Data.Foldable`.

We demonstrate as follows:

```
ghci> null' []
True
ghci> null' [1, 2, 3]
False
ghci> null' [1..]
False
```

We have the function `last'` with the following definition in `src/ch05/last.hs`:

5.6
```
import GHC.Stack (HasCallStack)

last' :: HasCallStack => [a] -> a
last' x = head (reverse x)
```

`last'` returns the last item in an argument with type that implements `HasCallStack`, an error if the argument is empty, or hangs indefinitely if the variable has infinite length.

We demonstrate as follows:

```
ghci> last' [1, 2, 3]
3
ghci> last' ["check", "mate"]
"mate"
ghci> last' []
*** Exception: Prelude.head: empty list
CallStack (from HasCallStack):
  error, called at libraries/base/GHC/List.hs:1646:3
    in base:GHC.List
```

```
  errorEmptyList, called at libraries/base/GHC/List.hs:85:11
    in base:GHC.List
  badHead, called at libraries/base/GHC/List.hs:81:28
    in base:GHC.List
  head, called at last.hs:4:11 in main:Main
  last', called at <interactive>:4:1 in interactive:Ghci3
```

We have the function `palindrome` with the following definition `src/ch05/palindrome.hs`:

5.7
```
import Distribution.Simple.Utils

palindrome :: String -> Bool
palindrome s = reverse (lowercase s) == lowercase s
```

`palindrome` uses the function `Distribution.Simple.Utils.lowercase` to check if the lowercase version of a string is the same as the lowercase version reversed, i.e., is the string a palindrome.

We demonstrate as follows:

```
ghci> palindrome "Radar"
True
ghci> palindrome "MadamImAdam"
True
ghci> palindrome "racecar"
True
ghci> palindrome "dog"
False
```

We find the first five elements of the infinite list $[9, 1, \ldots]$ as follows:

5.8
```
ghci> take 5 [9,1..]
[9,1,-7,-15,-23]
```

Thus we see that the first five elements are given by

$$[9, 1, \ldots] = [9, 1, -7, -15, -23, \ldots].$$

5.9 We have the function `cycle'` with the following definition in `src/ch05/cycle.hs`:

```
import GHC.Stack (HasCallStack)

cycle' :: forall a. HasCallStack => [a] -> [a]
cycle' xs = concat (repeat xs)
```

`cycle'` repeats an argument which implements `HasCallStack` an infinite number of times.

We demonstrate as follows:

```
ghci> take 10 (cycle' [4,7,8])
[4,7,8,4,7,8,4,7,8,4]
ghci> take 10 (cycle' [1])
[1,1,1,1,1,1,1,1,1,1]
```

5.10 (a) `["hello",42]` is not a valid expression, since it attempts to construct a list from elements of two different types, `String` and —Int—.

(b) `['h',"ello"]` is not a valid expression, since it attempts to construct a list from elements of two different types, `Char` and —String—.

(c) `['a','b','c']` is a valid expression.

(d) `length ['w','h','o']` is a valid expression.

(e) `length "hello"` is a valid expression.

(f) `reverse` is a valid expression, even though GHCI cannot print it.

5.11 It seems as if an arithmetic sequence will end at the last integer in the sequence before the last element in the constructor if the sequence is an integer sequence.

If it is a floating point sequence, i.e., one of the elements in the constructor was of floating point type, then the last number in the sequence will be the number in the sequence occurring after the last element in the constructor if that last element is further than the midpoint between two elements in the sequence, otherwise it will be the number occurring before.

We demonstrate as follows:

```
ghci> [0,3..7.5]
[0.0,3.0,6.0,9.0]
ghci> [0,3..7.49]
[0.0,3.0,6.0]
ghci> [0,3..7.499999999]
[0.0,3.0,6.0]
ghci> [0,3..7]
[0,3,6]
ghci> [0,3..8]
[0,3,6]
ghci> [0,3..9]
[0,3,6,9]
```

We have the following expression in `src/ch05/geometricseries.hs`:

5.12
```
series :: Double
series = sum [1.0 / n | n <- [1..100]]
```

used to calculate

$$\sum_{n=1}^{100} \frac{1}{n^2}.$$

Evaluating this in `ghci` results in the following:

```
ghci> series
5.187377517639621
```

We have the following expression in `src/ch05/factorial.hs`:

5.13
```
fact :: Integer -> Integer
fact n = product [1..n]
```

used to calculate $n!$. Evaluating this in `ghci` results in the following:

```
ghci> fact 1
1
ghci> fact 2
2
ghci> fact 3
6
ghci> fact 4
24
ghci> fact 5
120
```

We have the following set of functions in `src/ch05/exponential.hs`:

5.14
```
type R = Double

expList :: R -> [R]
expList x = [(1.0 + x / n) ** n | n <- [1 ..]]

expErr :: R -> R -> R
expErr x approx = abs (exp x - approx)

calcMinExpErr :: Int -> R -> R -> Int
calcMinExpErr n x eps approx =
  if expErr x (expList x !! n) < eps
    then n
    else calcMinExpErr (n + 1) x eps approx
s approx
```

In order to calculate how big n needs to be to get within 1 percent of the correct value for $x = 1$, we have the following:

```
ghci> calcMinExpErr 1 1 0.01
134
```

To calculate how big n needs to be to get within 1 percent of the correct value for $x = 10$, we have:

```
ghci> calcMinExpErr 1 10 0.01
```

This does not return a value within 10 minutes.

5.15 For this question, we update the code in `src/ch05/exponential.hs` to allow us to pass in a different approximation function, in this case the function `expSeries` and calculate the result. Thus we have

```
type R = Double

expList :: R -> [R]
expList x = [(1.0 + x / n) ** n | n <- [1 ..]]

expSeries :: R -> [R]
expSeries x = [(x ** n) / product [1 .. n] | n <- [1 ..]]

expErr :: R -> R -> R
expErr x approx = abs (exp x - approx)

calcMinExpErr :: Int -> R -> R -> (R -> [R]) -> Int
calcMinExpErr n x eps approx =
  if expErr x (approx x !! n) < eps
    then n
    else calcMinExpErr (n + 1) x eps approx
s approx
```

However, this does not return a result even for $x = 1$ within any reasonable time frame, so these functions must be optimized to produce a result more quickly.