Learn Physics with Functional Programming

—

Scott N. Walck

Graham Strickland

June 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

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

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

Thus we have an error of 1 percent if

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

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, $\operatorname{err}(x,\epsilon)$, compared to the exact derivative, $f'(x)$. Then, we have

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

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

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

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, $\mathrm{range}(2) = 0, 1, 2$, $\mathrm{range}(-4) = -4, -3, \ldots, 0$, and $\mathrm{range}(0) = 0$.

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.

6 Higher-Order Functions

6.1 We have the following function definitions in `src/Ch06/RockTrajectory.hs`

```
type R = Double

yRock :: R -> R -> R
yRock v0 t = v0 * t - 4.9 * t^2

vRock :: R -> R -> R
vRock v0 t = v0 - 9.8 * t
```

The first corresponds to the equation

$$y = v_0 t - \frac{1}{2} g t^2$$

and the second to

$$v = v_0 - g t.$$

6.2 We have the following

```
ghci> :t take 4
take 4 :: [a] -> [a]
```

We have the following

6.3
```
ghci> :t map
map :: (a -> b) -> [a] -> [b]
ghci> :t not
not :: Bool -> Bool
ghci> :t map not
map not :: [Bool] -> [Bool]
```

where `not` substitutes the type `Bool` for `a` and `b` in the definition of `map`, so that `map not` has the type `[Bool] -> [Bool]`.

6.4 We have the following definition in the file `src/Ch06/Geq.hs`

```
greaterThanOrEq7' :: Int -> Bool
greaterThanOrEq7' n = n >= 7
```

which we test with

```
ghci> greaterThanOrEq7' 10
True
ghci> greaterThanOrEq7' 7
True
ghci> greaterThanOrEq7' 5
False
```

We have the following definition in the file `src/Ch06/IntStrBool.hs`

6.5
```
import Data.Char

intStringBool :: Int -> String -> Bool
intStringBool n cs = case cs of
                [] -> False
                (c:cs) -> ord c == n || intStringBool n cs
```

The function `intStringBool` takes an `Int` n and returns a function that checks if any ASCII character in a string is equivalent to the number n and returns `True` if it is and `False` if it does not find a match. We demonstrate as follows

```
ghci> intStringBool 48 "Hello1"
False
ghci> intStringBool 49 "Hello1"
True
```

We have the following definition in the file src/Ch06/Predicate.hs

6.6
```
hasMoreThan6Elements :: [a] -> Bool
hasMoreThan6Elements xs = length xs > 6
```

which we test with

```
ghci> hasMoreThan6Elements "Hello, world!"
True
ghci> hasMoreThan6Elements "Hello"
False
ghci> hasMoreThan6Elements [1,2,3,4,5,6]
False
ghci> hasMoreThan6Elements [1,2,3,4,5,6,7]
True
```

The function `replicate` has the following definition

6.7
```
ghci> :t replicate
replicate :: Int -> a -> [a]
```

which indicates that it takes an input of type `Int` and an input of type `a` and produces a list of type `[a]`. The first three examples use types that are concatenated into lists. The fourth example uses the value `'x'` of type `Char`, which when concatenated together 3 times produces a list of `Char` which is simplified to the equivalent `String`, since a `String` is a list of `Char`.

6.8 We have the following definition in the file src/Ch06/Squares.hs

```
first1000Squares :: [Int]
first1000Squares = take 1000 [x ^ 2 | x <- [1..]]
```

which we test using

```
ghci> take 10 first1000Squares
[1,4,9,16,25,36,49,64,81,100]
```

We have the following definition in the file src/Ch06/Repeat.hs

6.9
```
repeat' :: a -> [a]
repeat' = iterate id
```

which we test using

```
ghci> take 10 (repeat' 'x')
"xxxxxxxxxx"
```

We have the following definition in the file src/Ch06/Replicate.hs

6.10
```
replicate' :: Int -> a -> [a]
replicate' n x = take n (repeat x)
```

which we test using

```
ghci> replicate' 3 'x'
"xxx"
```

We have the following test using GHCi

6.11
```
ghci> take 10 (iterate (\t -> t + 5) 0)
[0,5,10,15,20,25,30,35,40,45]
```

6.12 We have the following definition in the file `src/Ch06/Map.hs`

```
map' :: (a -> b) -> [a] -> [b]
map' fn as = [fn a | a <- as]
```

which we test using

```
ghci> map' sqrt [1,4,9]
[1.0,2.0,3.0]
```

We have the following definition in the file `src/Ch06/Filter.hs`

6.13
```
filter' :: (a -> Bool) -> [a] -> [a]
filter' predicate as = [a | a <- as, predicate a]
```

which we test using

```
ghci> filter' (\n -> n < 10) [6,4,8,13,7]
[6,4,8,7]
```

6.14 We have the following definition in the file `src/Ch06/Average.hs`

```
average :: [R] -> R
average xs = sum xs / fromIntegral (length xs)
```

which we test using

```
ghci> average [1.0, 2.0, 3.0]
2.0
```

Table 1 explains the two ways of thinking about the higher-order function `drop` and Table 2 provides the same for `replicate`:

Way of thinking	**Input to** `drop`	**Output from** `drop`
One-input thinking	`Int`	`[a] -> [a]`
Two-input thinking	`Int` and then `[a]`	`[a]`

Table 1: Two Ways of Thinking About the Higher-Order Function `drop`

Way of thinking	Input to `replicate`	Output from `replicate`
One-input thinking	`Int`	`a -> [a]`
Two-input thinking	`Int` and then `a`	`[a]`

Table 2: Two Ways of Thinking About the Higher-Order Function `replicate`

6.15 We have the following definition in the file `src/Ch06/TrapezoidalRule.hs`

```
trapSingle :: R -> (R -> R) -> R -> R
trapSingle dt fn x =
    let dx = dt / 4
     in sum [f * dx |
        f <- [
            0.5 * fn x,
            fn (x + dx),
            fn (x + 2 * dx),
            fn (x + 3 * dx),
            0.5 * fn (x + 4 * dx)
        ]
     ]

trapIntegrate :: Int        -- # of trapezoids n
              -> (R -> R)   -- function f
              -> R          -- lower limit a
              -> R          -- upper limit b
              -> R          -- result
trapIntegrate n f a b =
    let dt = (b - a) / fromIntegral n
     in sum [trapSingle dt f t | t <- [a, a + dt .. b - dt]]
```

which we test using the helper function `calcTrapIntegrateErr` as follows:

```
ghci> calcTrapIntegrateErr 1 (\x -> x ^ 3) 0 1 0.001
4
ghci> calcTrapIntegrateErr 1 (\x -> x ^ 3) 0 1e-6 0.001
2
ghci> calcTrapIntegrateErr 1 (\x -> exp (-x^2)) 0 1 0.001
3
```

7 Graphing Functions

7.1 We have the following definition in the file `src/Ch07/SinPlot.hs`

```
sinFunc :: R -> R
```

```
sinFunc = sin

sinPlot :: IO ()
sinPlot =
    plotFunc
        [Key (Just ["noautotitle"])]
        [-10, -9.9 .. 10]
        sinFunc
```

which we execute using

```
ghci> :l
Ok, no modules loaded.
ghci> :m
ghci> :l Ch07.SinPlot
[1 of 1] Compiling Ch07.SinPlot
( src/Ch07/SinPlot.hs, interpreted )
Ok, one module loaded.
ghci> sinPlot
```

to produce the plot in Figure 1.

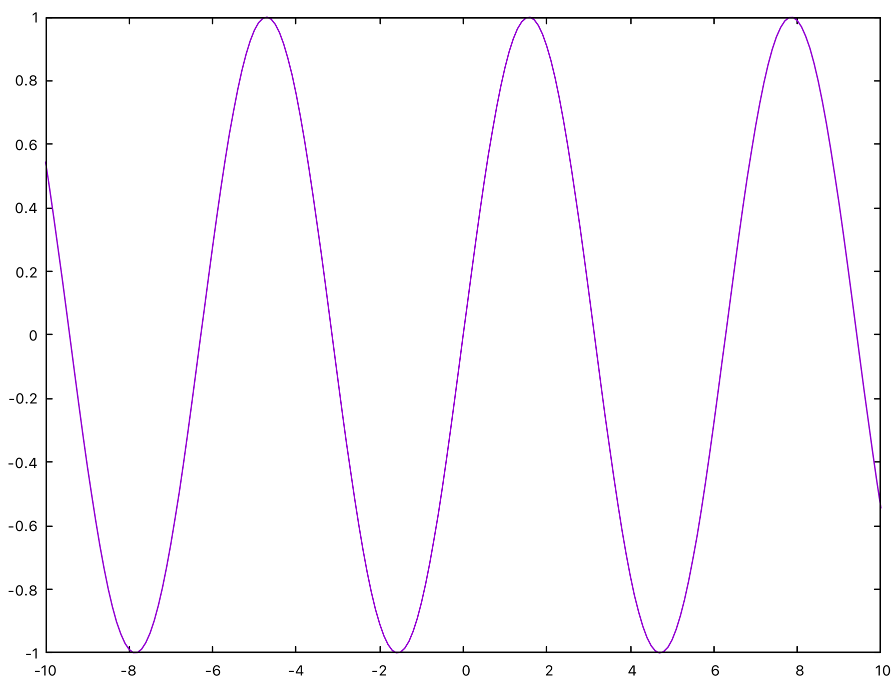

Figure 1: Plot of $sin(x)$ from $x = -10$ to $x = 10$

7.2 We have the following definition in the file `src/Ch07/PlotYRock30.hs`

```
yRock30 :: R -> R
yRock30 t = 30 * t - 0.5 * 9.8 * t ** 2

yRock30Plot :: IO ()
yRock30Plot =
    plotFunc
        [Key (Just ["noautotitle"])]
        [0, 0.1 .. 6]
        yRock30
```

from which we produce the plot in Figure 2.

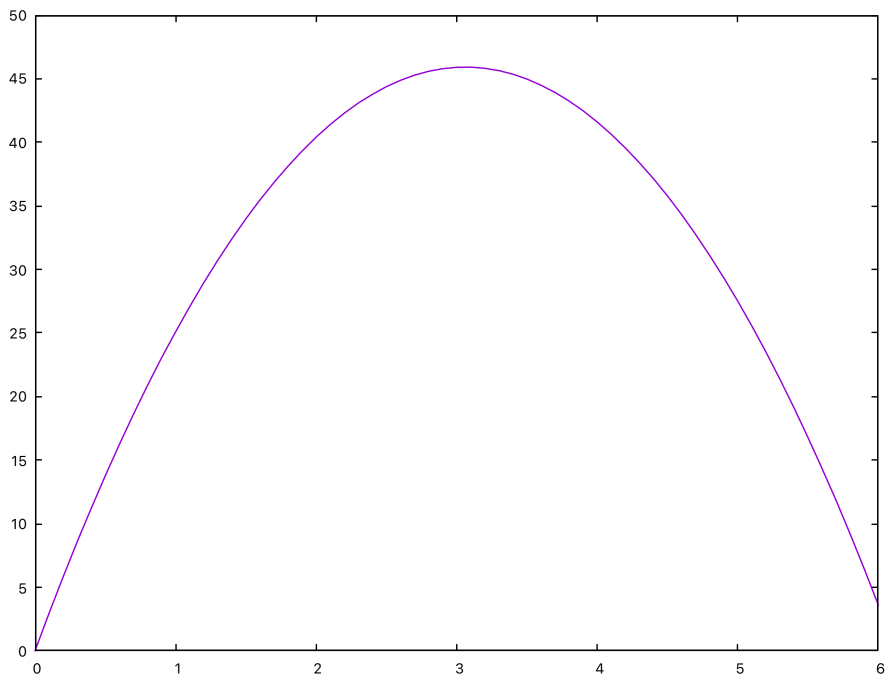

Figure 2: Plot of `yRock30` function from $t = 0$ to $t = 6s$

7.3 We have the following definition in the file `src/Ch07/PlotYRock30.hs`

```
yRock :: R -> R -> R
yRock v0 t = v0 * t - 0.5 * 9.8 * t ** 2

yRock20Plot :: IO ()
yRock20Plot =
```

```
plotFunc
    [Key (Just ["noautotitle"])]
    [0, 0.1 .. 4]
    (yRock 20)
```

from which we produce the plot in Figure 3.

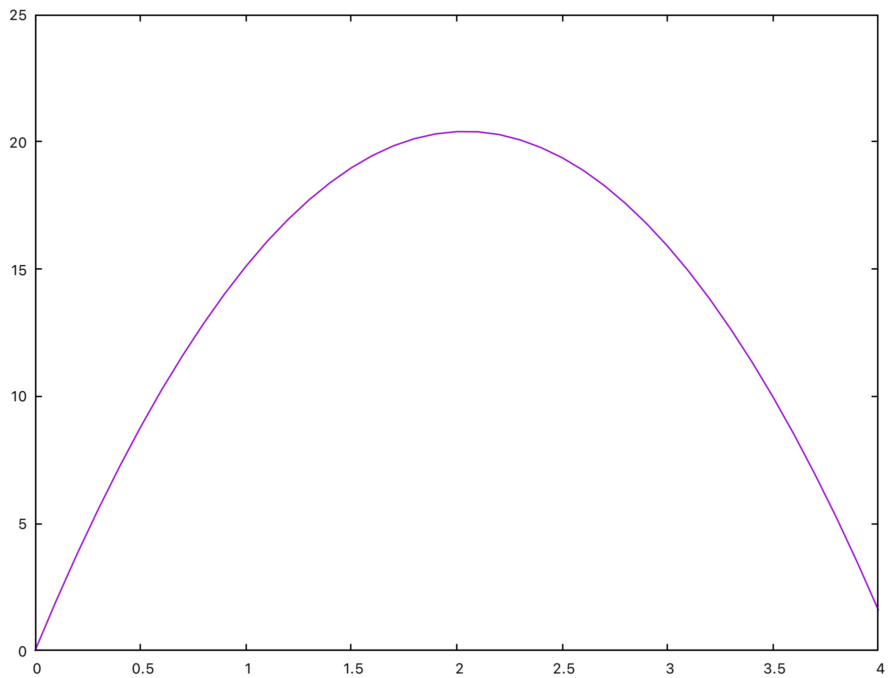

Figure 3: Plot of `yRock 20` function from $t = 0$ to $t = 4s$

8 Type Classes

8.1 Yes, for example, the type `Float` belongs to both `Floating` and `Fractional`.

8.2 (a) Two functions f and g are equal if and only if they have the same domain and range, and for every element of the domain, they map to the same element of the range, i.e., if $f : X \to Y$ and $g : X \to Y$, then

$$\begin{aligned} & f = g \\ \Leftrightarrow\ & f(x) = g(x) \forall x \in X. \end{aligned}$$

(b) The computer cannot necessarily compute each element of the domain in a reasonable time to ensure that both functions map each element of the domain to the same element in the range.

(c)
```
f :: Bool a => a -> a
```

8.3 No, the function `(/2)` computes division by 2, while the function `(2/)` computes division by 2 of another argument, e.g., $1/2 \neq 2/1$.

8.4 Depending upon which type of base we are squaring, we could use the sections `(^2)`, `(^^)`, or `(**)`.

8.5 (a) We have the following from GHCi:

```
ghci> :i Integer
type Integer :: *
data Integer
  = GHC.Num.Integer.IS ghc-prim-0.9.1:GHC.Prim.Int#
  | GHC.Num.Integer.IP ghc-prim-0.9.1:GHC.Prim.ByteArray#
  | GHC.Num.Integer.IN ghc-prim-0.9.1:GHC.Prim.ByteArray#
        -- Defined in ‘GHC.Num.Integer’
instance Integral Integer -- Defined in ‘GHC.Real’
instance Num Integer -- Defined in ‘GHC.Num’
instance Real Integer -- Defined in ‘GHC.Real’
instance Enum Integer -- Defined in ‘GHC.Enum’
instance Eq Integer -- Defined in ‘GHC.Num.Integer’
instance Ord Integer -- Defined in ‘GHC.Num.Integer’
instance Read Integer -- Defined in ‘GHC.Read’
instance Show Integer -- Defined in ‘GHC.Show’
```

Thus we can see that `Integer` also belongs to `Real`, indicating that for an integer $x \in \mathbb{Z}$, we have $x \in \mathbb{R}$, since $\mathbb{Z} \subseteq \mathbb{R}$. We also see that `Integer` belongs to `Enum`, which represents the set of enumerable (countable) numbers. Finally, `Integer` also belongs to `Read`, which is the input type corresponding to `Show`, indicating that elements of `Read` can be serialized.

(b) Executing the GHCi command `:info` yields:

```
ghci> :i Enum
type Enum :: * -> Constraint
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

```
  {-# MINIMAL toEnum, fromEnum #-}
        -- Defined in ‘GHC.Enum’
instance Enum Double -- Defined in ‘GHC.Float’
instance Enum Float -- Defined in ‘GHC.Float’
instance Enum () -- Defined in ‘GHC.Enum’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Enum Char -- Defined in ‘GHC.Enum’
instance Enum Int -- Defined in ‘GHC.Enum’
instance Enum Integer -- Defined in ‘GHC.Enum’
instance Enum Ordering -- Defined in ‘GHC.Enum’
instance Enum a => Enum (Solo a) -- Defined in ‘GHC.Enum’
instance Enum Word -- Defined in ‘GHC.Enum’
```

Thus we see that `Double`, `Float`, `()`, `Bool`, `Char`, `Int`, `Integer`, `Ordering`, `Solo`, and `Word` are instances of `Enum`.

8.6 (a) `42 :: Num a => a`
(b) `42.0 :: Fractional a => a`
(c) `42.5 :: Fractional a => a`
(d) `pi :: Floating a => a`
(e) `[3,1,4] :: Num a => [a]`
(f) `[3,3.5,4] :: Fractional a => [a]`
(g) `[3,3.1,pi] :: Floating a => [a]`
(h) `(==) :: Eq a => a -> a -> Bool`
(i) `(/=) :: Eq a => a -> a -> Bool`
(j) `(<) :: Ord a => a -> a -> Bool`
(k) `(<=) :: Ord a => a -> a -> Bool`
(l) `(+) :: Num a => a -> a -> a`
(m) `(-) :: Num a => a -> a -> a`
(n) `(*) :: Num a => a -> a -> a`
(o) `(/) :: Fractional a => a -> a -> a`
(p) `(^) :: (Num a, Integral b) => a -> b -> a`
(q) `(**) :: Floating a => a -> a -> a`
(r) `8/4 :: Fractional a => a`
(s) `sqrt :: Floating a => a -> a`
(t) `cos :: Floating a => a -> a`
(u) `show :: Show a => a -> String`
(v) `(2/) :: Fractional a => a -> a`

8.7 Because the operator `/` has type `(/) :: Fractional a => a -> a -> a`.

8.8 Using the helper functions in `src/Ch08/Quotients.hs`, we see that

```
ghci> quot (-4) 3 == div (-4) 3
False
```

and

```
ghci> quot (-4) 3
-1
ghci> div (-4) 3
-2
```

Thus we can conclude that for integer division with negative numerators, `quot` rounds down to the next integer in the positive direction, while `div` rounds down in the negative direction.

Likewise, we see that

```
ghci> rem (-4) 3 == mod (-4) 3
False
```

and

```
ghci> rem (-4) 3
-1
ghci> mod (-4) 3
2
```

Thus we conclude that when calculating remainders with negative numerators, `rem` returns the negative of the remainder when the modulus of the numerator is divided by the denominator, while `mod` calculates a where $a \equiv m (\bmod n)$, for m the argument on the left and n that on the right.

8.9 Table 3 shows which types can be used for the base `x` and the exponent `y` in the expresion `x ^ y` while Table 4 does the same for the expression `x ** y`.

	`y :: Int`	`y :: Integer`	`y :: Float`	`y :: Double`
`x :: Int`	^	^		
`x :: Integer`	^	^		
`x :: Float`	^	^		
`x :: Double`	^	^		

Table 3: Possible Types for `x` and `y` with the Single-Caret Exponentiation Operator

We can see from all three tables that there is no applicable exponentiation operator if the base has type `Float` and the exponent type `Double`.

	`y :: Int`	`y :: Integer`	`y :: Float`	`y :: Double`
`x :: Int`				
`x :: Integer`				
`x :: Float`			`**`	
`x :: Double`				`**`

Table 4: Possible Types for `x` and `y` with the Double-Asterisk Exponentiation Operator