Learn Physics with Functional Programming
—
Scott N. Walck

Graham Strickland

February 22, 2026

4 Describing Motion

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

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

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` defined in `src/Ch05/Range.hs`.

`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 0
[0]
```

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 definition in `src/Ch05/Cycle.hs`.

`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 define the expression in `src/Ch05/GeometricSeries.hs` used to calculate

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

Evaluating this in `ghci` results in the following:

5.12
```
ghci> series
5.187377517639621
```

5.14 We have the definitions of these functions in `src/Ch05/Exponential.hs`.

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 define the functions in `src/Ch06/RockTrajectory.hs`.

The first corresponds to the equation

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

and the second to

$$v = v_0 - gt.$$

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]`.

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

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
```

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`.

6.11 We have the following test using GHCi

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

Table 6.1 explains the two ways of thinking about the higher-order function `drop` and Table 6.2 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 6.1: 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 6.2: Two Ways of Thinking About the Higher-Order Function `replicate`

6.16 Using the definition in the file `src/Ch06/TrapezoidalRule.hs`, which we test using the helper function `calcTrapIntegrateErr`, we have

```
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 Using the definition in the file `src/Ch07/SinPlot.hs`, we have

```
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
```

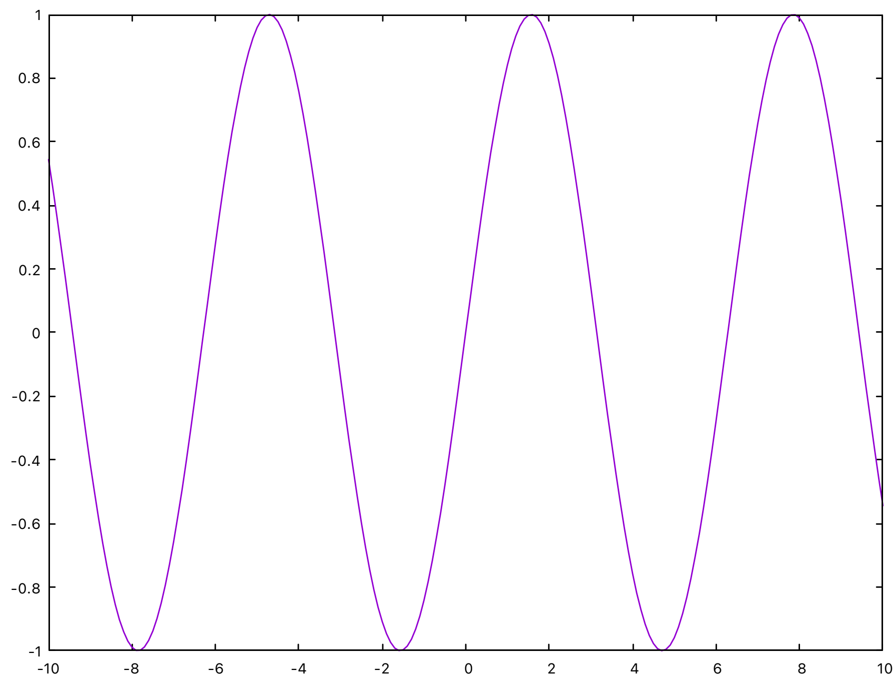

Figure 7.1: Plot of $sin(x)$ from $x = -10$ to $x = 10$

to produce the plot in Figure 7.1.

7.2 Using the definition in the file `src/Ch07/PlotYRock30.hs`, we produce the plot in Figure 7.2.

7.3 Using the definition in the file `src/Ch07/PlotYRock30.hs`, we produce the plot in Figure 7.3.

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

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

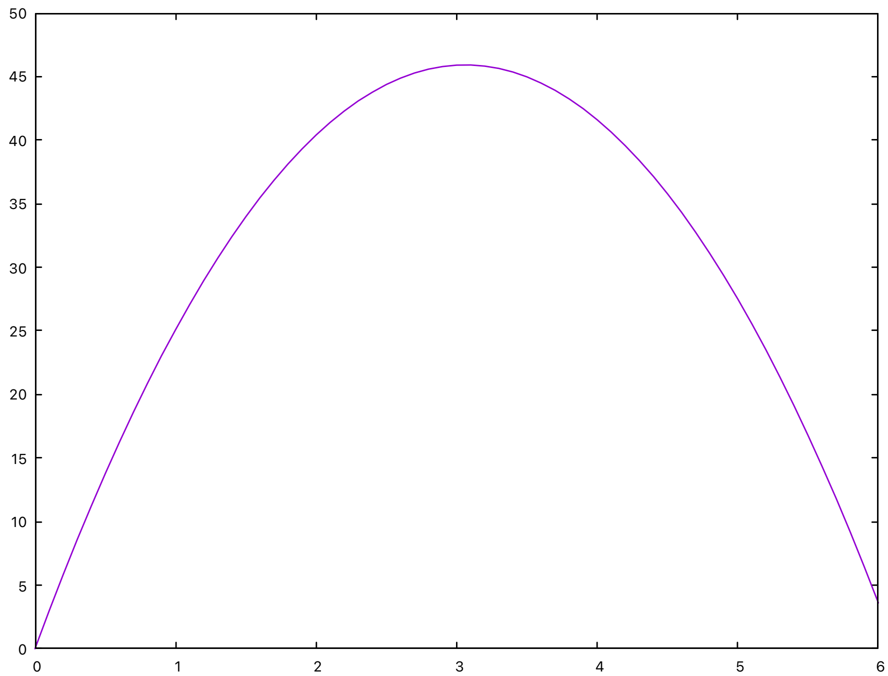

Figure 7.2: Plot of `yRock30` function from $t = 0$ to $t = 6s$

(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’
```

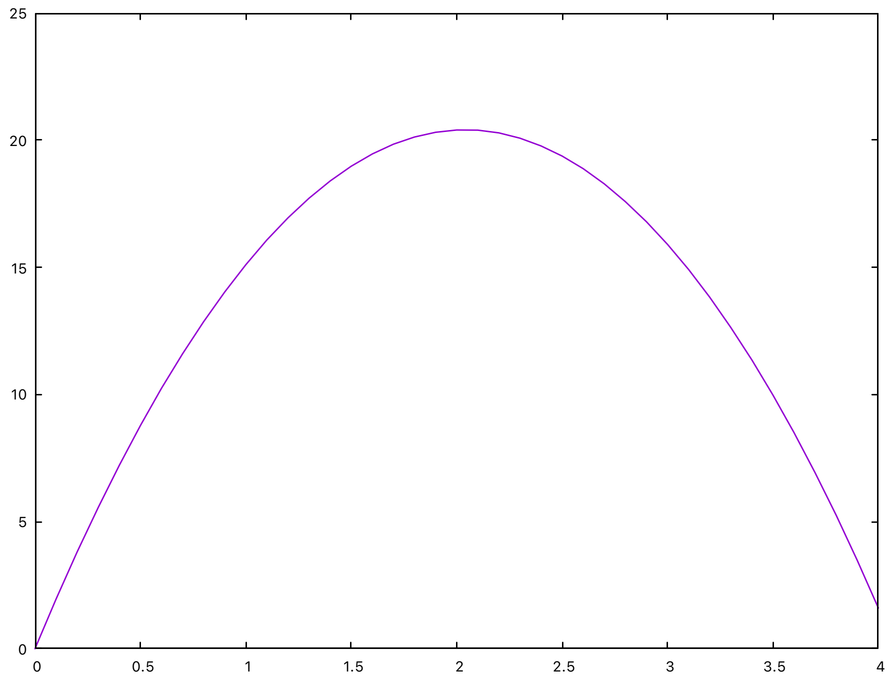

Figure 7.3: Plot of `yRock20` function from $t = 0$ to $t = 4s$

```
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 $a \equiv m (\mathrm{mod}\, n)$, for m the argument on the left and n that on the right.

8.9 Table 8.1 shows which types can be used for the base `x` and the exponent `y` in the expresion `x ^ y` while Table 8.2 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 8.1: 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`.

9 Tuples and Type Constructors

9.2 `curry` takes as input a function which takes a tuple `(a, b)` as input and outputs a value of type `c` and returns a function which takes in two variables `a` and `b` and outputs a value `c`. `uncurry` does the opposite.

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

Table 8.2: Possible Types for `x` and `y` with the Double-Asterisk Exponentiation Operator

9.5 The remaining elements of the longer list are discarded, e.g.,

```
ghci> zip [1] [1,2]
[(1,1)]
```

No, for instance, we have the following error when trying to convert a list of empty tuples (the list containing the unit type):

9.7
```
ghci> (zip' . unzip) [()]

<interactive>:10:17: error:
    • Couldn't match expected type '(a, b)' with actual type '()'
    • In the expression: ()
      In the first argument of 'zip' . unzip', namely '[()]'
      In the expression: (zip' . unzip) [()]
    • Relevant bindings include
        it :: [(a, b)] (bound at <interactive>:10:1)
```

Yes, since this does return exactly the expression it was passed in, e.g.,

```
ghci> (unzip . zip') ([], [])
([],[])
ghci> (unzip . zip') ([1,2,3], ["Albert","Isaac","James"])
([1,2,3],["Albert","Isaac","James"])
```

We have the following:

9.8
```
ghci> lookup "Albert Einstein" grades
Just 89
ghci> lookup "James Clerk Maxwell" grades
Nothing
```

9.14 Using the definitions from `src/Ch09/RockTrajectory.hs`, we produce the plot in Figure 9.1.

10 Describing Motion in Three Dimensions

10.1 (a)
```
v0 :: Vec
v0 = 20 *^ iHat
```
(b)
```
v1 :: Vec
v1 = 20 *^ iHat ^-^ 9.8 *^ kHat
```
(c)
```
v :: R -> Vec
v t = 20 *^ iHat ^-^ 9.8 *^ t *^ kHat
```

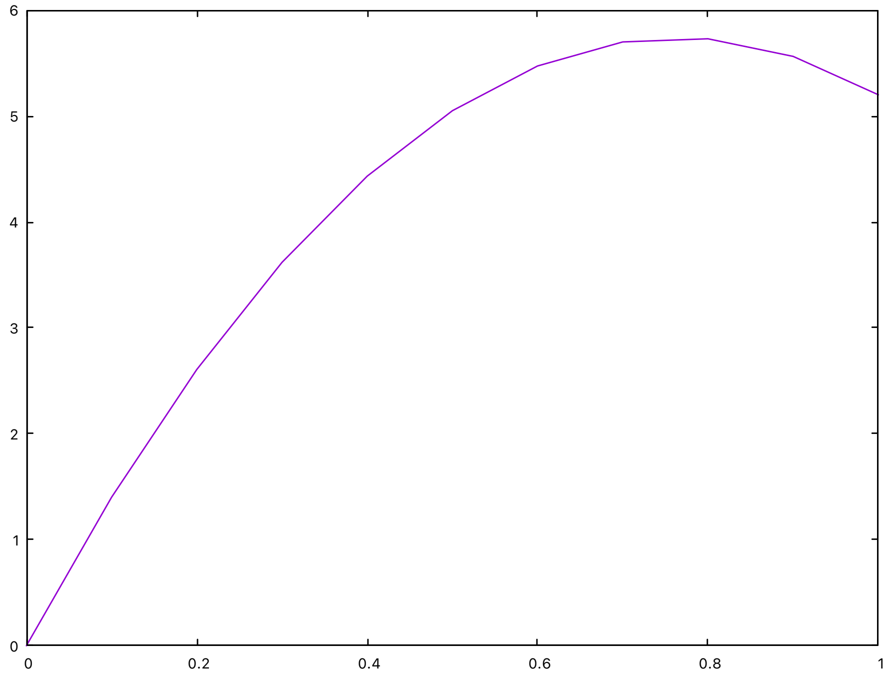

Figure 9.1: Plot of t and y values for `rockTrajectory`

(d)
```
r :: R -> Vec
r t = 30 *^ jHat ^+^ 20 *^ t *^ iHat ^-^ 4.9 *^ t ** 2 *^ kHat
```

(e)
```
x :: R -> R
x t = iHat <.> r t
```

10.2 Using the definition from `src/Ch10/VecIntegral.hs`, we have

```
ghci> vecIntegral 0.01 v1 0 10
vec 6.666665000000005 0.7499996250000007 1.9999983333336274e-2
```

which gives an approximation to the integral from 0 to 10 with respect to time for the vector function

$$\mathbf{v}_1(t) = 2t^2\hat{\mathbf{i}} + 3t^3\hat{\mathbf{j}} + t^4\hat{\mathbf{k}}.$$

10.3 Using the definition from `src/Ch10/MaxHeight.hs`, we have

```
ghci> maxHeight (vec 1.0 1.0 1.0) (5.0 *^ kHat)
2.2737499999999997
```

Using the definition from `src/Ch10/Speed.hs`, we have

10.4
```
ghci> speedCA (Vec 1.0 1.0 1.0) (9.81 *^ negateV kHat) 1.0
8.922785439536243
```

10.5 Using the definition from `src/Ch10/ProjectileVel.hs`, we have

```
ghci> projectileVel (Vec 1.0 1.0 1.0) 1.0
vec 1.0 1.0 (-8.81)
ghci> projectileVel (Vec 1.0 1.0 1.0) 2.0
vec 1.0 1.0 (-18.62)
```

The file `src/Ch10/Vec2D.hs` contains the definition of the `Vec2D` class. We demonstrate the use of `magAngleFromVec2D` and `vec2DFromMagAngle` using

10.6
```
ghci> magAngleFromVec2D jHat
(1.0,1.5707963267948966)
ghci> magAngleFromVec2D iHat
(1.0,0.0)
ghci> magAngleFromVec2D (negateV iHat)
(1.0,-3.141592653589793)
ghci> magAngleFromVec2D (negateV jHat)
(1.0,-1.5707963267948966)
ghci> vec2DFromMagAngle (1.0,-1.5707963267948966)
vec 6.123233995736766e-17 (-1.0)
ghci> vec2DFromMagAngle (1.0,0.0)
vec 1.0 0.0
ghci> vec2DFromMagAngle (1.0,0.0)
```

10.9 Using the definition in the file `src/Ch10/BallSpeed.hs`, we have plot the z component of the ball's trajectory in Figure 10.1 and the rate of change of its speed in Figure 10.2.

From this plot, we can tell that the rate of change of its speed is equal when it reachs time $t = 4$s. At this point its acceleration is, by definition $a = 0$, while its velocity is given by $v \approx 61.56745393019432$.

10.11 We have the following output using the functions defined in `src/Ch10/NonuniformCircularMotion.hs`

```
ghci> radialComponent 2 2
vec (-69.11249319950134) 20.10935225661961 0.0
ghci> radialComponent 2 2
ghci> vNCM 0.01 (2, thetaFunc) 2
11.999999999999744
ghci> magRadialComponent 2 2
71.9786271363377
ghci> squareSpeedDivRad 2 2
71.99999999999693
```

Thus we can conclude that the radial component of acceleration at $t = 2$s is given by the vector

$$\mathbf{a}_r\left(\theta(2), 2\right) \approx (-69.11249319950134, 20.10935225661961, 0.0),$$

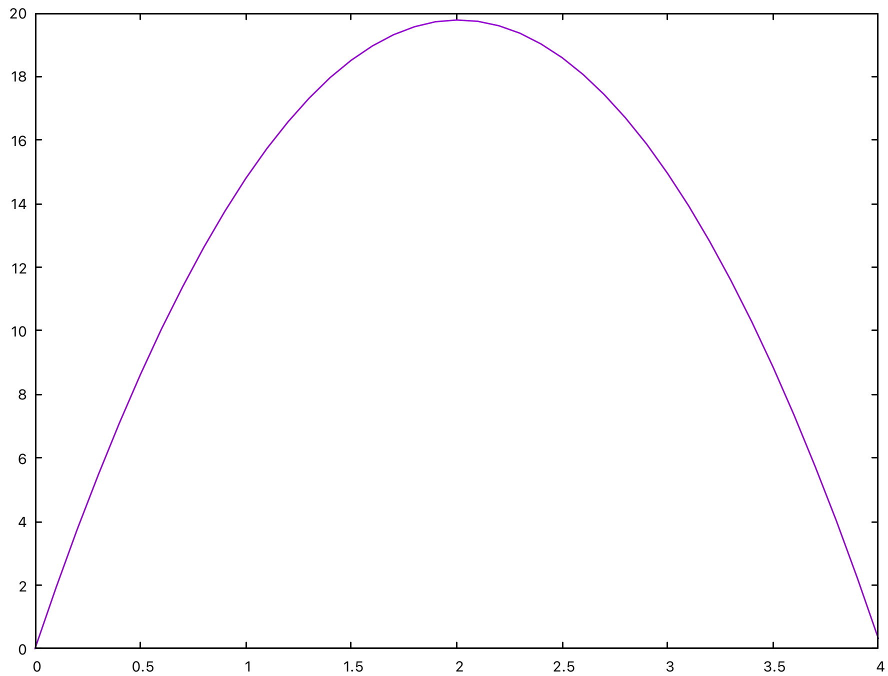

Figure 10.1: Plot of the z component of the ball's trajectory as a function of time

the speed of the particle at time $t = 2$s is given by

$$v_{\mathrm{NCM}}\left(\theta(2), 2\right) \approx 11.999999999999744,$$

and the magnitude of the radial component at that time by

$$\left|\mathbf{a}_r\left(\theta(2), 2\right)\right| \approx 71.99999999999693.$$

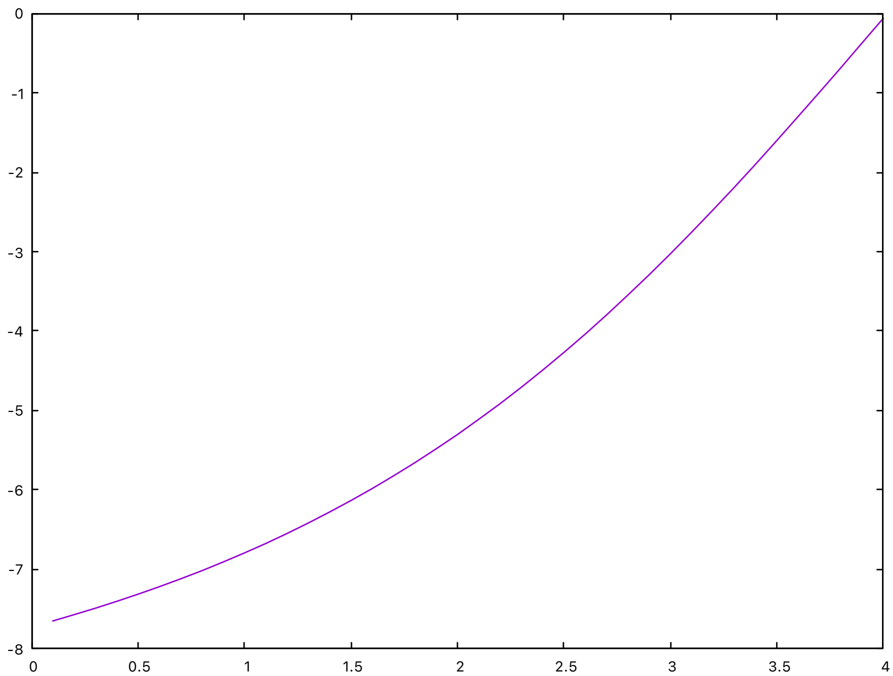

Figure 10.2: Plot of rate of change of the speed of the ball as a function of time