14

NEWTON'S SECOND LAW

Isaac Newton accomplished a lot. Among the numerous physical and mathematical insights he left us are three numbered laws that bear his name. Newton's second law is the most important of these; it provides a method for understanding the motion of an object if we know the forces that act on it. Newton's third law is almost as significant; it's a rule about how two objects interact. Newton's first law, from a mathematical standpoint, is a corollary to Newton's second law, so it seems the smallest of the three. But since Newton's second law is sufficiently intuition shattering, it's helpful to get our minds around something simpler before trying to grasp it. Newton's first law serves well in this capacity; it makes a bold claim that seems obviously false.

In this chapter, we'll discuss Newton's first law and then turn our attention to Newton's second law in one linear dimension, such as a horizontal line or a vertical line. We'll show how to think about Newton's second law

in a sequence of settings of increasing complexity, organized by what the forces depend on. We'll start with constant forces, the simplest situation, before moving on to forces that depend only on time. Then we'll turn to forces that depend on the velocity of the particle they act on, followed by forces that depend on both time and velocity. The techniques for solving Newton's second law change as the forces involved depend on different physical quantities. We'll introduce the Euler method for solving a differential equation and explore a number of situations in which Newton's second law is the central principle that allows us some traction in understanding the motion of an object.

Newton's First Law

Let's return to the air track of Chapter 4. If you give the car a little push on the air track and then let it go, it will travel at a constant speed until it hits the end of the track. After we stop pushing the car, it continues to move at some speed even with no force applied in the direction of motion. This tendency for moving objects to keep moving is called *inertia*. The idea of inertia is relevant in the one-dimensional spatial setting of the air track, and it's also relevant in the unconstrained three-dimensional spatial setting of the world in which we live. The idea is important enough to be enshrined in a principle of physics called *Newton's first law*. Here are three versions:

> **Newton's first law, Newton's words [15]**
> *Every body perseveres in its state of being at rest or of moving uniformly straight forward, except insofar as it is compelled to change its state by forces impressed.*
>
> **Newton's first law, poetic version**
> *A body in motion stays in motion. A body at rest stays at rest.*
>
> **Newton's first law, modern version**
> *In the absence of applied forces, an object maintains the same velocity.*

Recall that velocity is a vector, so maintaining the same velocity means keeping the same speed as well as the same direction. Since acceleration is change in velocity per unit of time, an equivalent way of expressing Newton's first law is that in the absence of forces, an object experiences no acceleration.

Notice that Newton's first law makes no mention of forces that were applied *in the past*. The point is that if there are no forces acting *now*, the velocity will stay constant now. Any time there are no forces present, the velocity will stay constant.

Why does Newton's first law seem obviously false? Because we're stuck on the surface of the earth, a place that is rife with forces we might fail to consider, friction and air resistance not least among them. Things are a bit simpler out in space. We can imagine one astronaut tossing a small wrench

to another at slow speed. The wrench just glides straight across the ship, perhaps rotating slowly about its center. That wrench is a great example of Newton's first law.

Perhaps you've been in a car when the driver slams on the brakes so that books, papers, and toys go flying forward (with respect to the car's seats). In my family, we celebrate these moments by shouting "Newton's first law!" From a perspective outside the (decelerating) car, the books, papers, and toys are doing their best to travel in a straight line, at least for the short period of time before gravity and other objects put an end to their line-like motion.

Newton's first law tells us that objects naturally go steady and straight. In practice, though, they don't. Newton's second law explains how and why.

Newton's Second Law in One Dimension

Newton's first law tells us that when no forces are present, an object does not accelerate. Newton's second law claims that acceleration is caused by forces.

> **Newton's second law, Newton's words [15]**
> *A change in motion is proportional to the motive force impressed and takes place along the straight line in which that force is impressed.*
>
> **Newton's second law, poetic version**
> *An object's acceleration is directly proportional to the net force acting on the object and inversely proportional to its mass.*

Modern versions of Newton's second law are expressed by Equation 14.1 for Newton's second law in one dimension, and Equation 16.1 for Newton's second law in three dimensions. In the remainder of this chapter, we'll treat Newton's second law in one dimension, which allows us to keep things simple by using numbers rather than vectors for velocity, acceleration, and force. In Chapter 16, we'll treat Newton's second law in full generality with vectors.

To discuss force and mass in a quantitative way, we need units of measure. In the SI system, force is measured in Newtons (N). A 100-N force has a different effect on a golf ball than it has on a bowling ball. According to Newton, each object has a *mass*, which determines the readiness of an object to accelerate in response to a force. A large-mass object experiences small acceleration compared to a small-mass object exposed to the same force. The SI unit of mass is the kilogram (kg).

Newton's second law expresses a relationship between the following three quantities:

- The forces that act on an object
- The mass of the object
- The acceleration of the object

Newton's second law says that the acceleration of an object can be found by dividing the net force acting on the object by the mass of the object. The *net force* acting on an object is the sum of all the forces acting on the object. In one dimension, some forces may be negative and some may be positive.

Newton's second law is usually written as $F_{\text{net}} = ma$. Unlike the one-dimensional equations for velocity and acceleration (Equations 4.5 and 4.12), this equation is not an equality of functions. The acceleration of the object is only a function of time, but the net force generally depends on the time, the position of the object, and the velocity of the object. The net force at time t is $F_{\text{net}}(t, x(t), v(t))$. A better way to write Newton's second law is:

Newton's second law in one dimension

$$F_{\text{net}}(t, x(t), v(t)) = ma(t) \tag{14.1}$$

There is a chicken-and-egg issue going on with Newton's second law. We know from Equations 4.5 and 4.12 that $v = Dx$ and $a = Dv$. If we know the function a (meaning we know its value at all times), we can find the function v given an initial velocity. (See Equation 6.1 and the corresponding function `velFromAcc`.) We can then go on and find the function x given an initial position. (See Equation 6.5 and the corresponding function `posFromVel`.) But Newton's second law is telling us that acceleration depends on the forces, which depend on the position and the velocity. To find the position of my object, it seems that I need to find the velocity, and for that I need the acceleration. However, the acceleration depends on both the position and the velocity.

There is a name for this particular kind of chicken-and-egg problem. Newton's second law is an example of a *differential equation*. A differential equation is a relationship between derivatives of an unknown function, with the unknown function itself often regarded as the zeroth derivative. The unknown function in the case of Newton's second law is usually either the position x or the velocity v. Velocity can be written as the first derivative of position ($v = Dx$), and acceleration can be written as the second derivative of position ($a = Dv = D^2x$).

Newton's second law looks more like a differential equation if we write it in terms of an unknown position function.

$$F_{\text{net}}(t, x(t), Dx(t)) = mD^2x(t) \tag{14.2}$$

This is a second-order differential equation because it is a relationship between the position function x, its first derivative Dx, and its second derivative D^2x. The relationship for a particular physical object depends on the function F_{net}, which depends on the nature of the forces acting on the object.

In simple situations, the net force on an object may not depend on time, position, and velocity, but rather only on zero, one, or two of these physical quantities. In these simple situations, Newton's second law may appear as something simpler than a second-order differential equation. Table 14-1 lists situations by the physical quantities that the forces depend on and gives the mathematical technique needed to solve Newton's second law.

Table 14-1: The Technique for Solving Newton's Second Law, Based on Which Physical Quantities the Forces Depend On

Forces depend only on	Solution technique
Nothing	Algebra
Time	Integration
Velocity	First-order differential equation
Time and velocity	First-order differential equation
Time, position, and velocity	Second-order differential equation

A net force that depends on nothing is a constant net force. Its value remains constant over time, independent of time, position, or velocity. In the next several sections, we'll look at constant forces, forces that depend only on time, forces that depend only on velocity, and forces that depend on both time and velocity. This restriction allows us to limit our attention in this chapter to first-order differential equations. In Chapter 15, we'll look at the more general case of one-dimensional motion in which the net force can depend on time, position, *and* velocity.

Second Law with Constant Forces

The simplest situation for Newton's second law is when the net force is constant, independent of time, position, and velocity. Most problems in an introductory physics course are like this because they can be solved without differential equations and without a computer.

Let's consider an example problem with constant forces.

Example 14.1. Suppose we have a car with mass 0.1 kg on an air track. The car is initially moving east at a speed of 0.6 m/s. Starting at time $t = 0$, we apply to this car a constant force of 0.04 N to the east. At the same time, our friend applies to the same car a constant force of 0.08 N to the west. What will the subsequent motion of the car look like? In particular, how will the velocity and the position of the car change in time?

Figure 14-1 shows the schematic diagram.

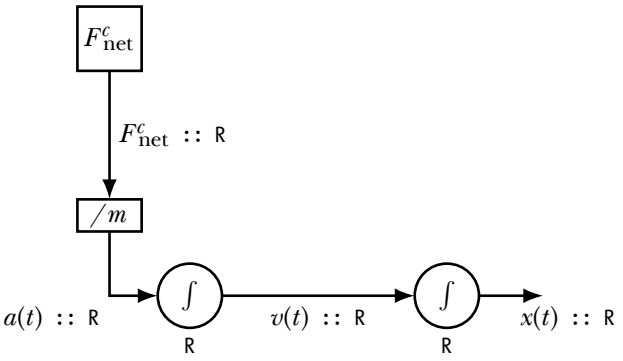

Figure 14-1: Schematic diagram for Newton's second law with constant forces

The constant net force F^c_{net} (superscript c for constant) acting on the object needs to be divided by the mass of the object to obtain the acceleration of the object. Because the net force is constant, the acceleration is also constant.

$$a(t) = \frac{F^c_{\text{net}}}{m} \tag{14.3}$$

We write $a(t)$ rather than a for acceleration, not because acceleration changes with time, but because a is the acceleration function (type `R -> R`) and $a(t)$ is the acceleration (type `R`). We then integrate acceleration to obtain the velocity.

$$v(t) = v(0) + \frac{F^c_{\text{net}} t}{m} \tag{14.4}$$

The integrator that produces velocity contains a real number (type `R`) as state. This type is shown below the integrator in Figure 14-1. This integrator remembers the current velocity so that it can be updated using the acceleration.

We then integrate the velocity to obtain the position.

$$x(t) = x(0) + v(0)t + \frac{F^c_{\text{net}} t^2}{2m} \tag{14.5}$$

The wires of the diagram represent quantities that are continuously changing in time. Each wire in the diagram is labeled with a name and a type. For this diagram, all of the wire types are real numbers.

Rectangular boxes represent purely functional constants and functions. In other words, they are constants and functions that do not contain any state, so that the output is a function only of the input. The circular integrators contain states that must be combined with the input to produce the output. The integrators are labeled with the type of state they contain, which is the same as the type of the output from the integrator.

Before we write Haskell code to solve Newton's second law for constant forces, we are going to write a few lines of code that need to be at the top of the source code file we build throughout this chapter. The first line turns on warnings, which I recommend doing because the compiler will warn you of things that are legal but unusual enough that they may not be what you intended. The second line gives the code in this chapter the module name `Newton2`. If we want to use functions we write here in later chapters, we'll refer to the current code using its module name. A module name is optional, but if you use one, it must match the filename; in this case, the filename should be *Newton2.hs*. The third line loads the `gnuplot` graphics library so that we can make a graph. Imports like this must occur before any function definitions or type signatures.

```
{-# OPTIONS -Wall #-}

module Newton2 where

import Graphics.Gnuplot.Simple
```

Example 14.1 is typical of situations in which Newton's second law applies. Given a mass, an initial velocity, and some forces, we are asked to produce velocity as a function of time. In the Haskell language, a solution to this example situation would be a (higher-order) function `velocityCF` (`CF` for constant forces) with the following type:

```
velocityCF :: Mass
           -> Velocity          -- initial velocity
           -> [Force]           -- list of forces
           -> Time -> Velocity  -- velocity function
```

Recall that there are (at least) two ways to read this type signature. On one reading, `velocityCF` takes four inputs—mass, initial velocity, a list of forces, and a time—and produces as output a real number representing velocity. An alternative reading is that `velocityCF` takes three inputs—mass, initial velocity, and a list of forces—and produces as output a *function* for how velocity changes with time. If we wanted to emphasize the latter viewpoint, we could write

```
velocityCF :: Mass -> Velocity -> [Force] -> (Time -> Velocity)
```

but it means the same thing as the original type signature.

We used the types `Time`, `Mass`, `Velocity`, and `Force`. These are not built-in types in Haskell, so we'd better define what they mean. In one-dimensional mechanics, all of these quantities can be represented with real numbers, so we can write some type synonyms to define these types. Using a type synonym in which `R` stands for `Double`,

```
type R = Double
```

we can write type synonyms for all of the other types:

```
type Mass     = R
type Time     = R
type Position = R
type Velocity = R
type Force    = R
```

The definitions for types `Mass`, `Time`, and so on, need not appear before their use in a type signature. Haskell allows definitions of constants, functions, and types before or after their use.

If we can write a function `velocityCF` with the type signature above, we will have solved not just Example 14.1, but all others like it. Our strategy in writing such a function is:

- Find the net force by adding all of the forces
- Find the acceleration using Newton's second law (Equation 14.3)
- Find the velocity from the acceleration (Equation 4.14 or 14.4)

Here's a definition for `velocityCF` that expresses these three steps and has the type we claimed earlier.

```
velocityCF m v0 fs
    = let fNet = sum fs       -- net force
          a0   = fNet / m     -- Newton's second law
          v t  = v0 + a0 * t  -- constant acceleration eqn
      in v
```

To write the function `velocityCF`, we begin by naming the three inputs: mass `m`, initial velocity `v0`, and list of forces `fs`. We then use a `let` construction to define three local names for net force, acceleration, and velocity. To find the net force, we sum up the forces in the list using the built-in `sum` function. To find the acceleration, we divide the net force on the object by the mass of the object, as Newton's second law prescribes.

The third equation in the `let` construction defines a local function `v` to represent the velocity function. We use Equation 4.14, one of the constant acceleration equations introduced in standard introductory physics textbooks, but we could just as easily have used Equation 14.4 in place of the second and third lines of the `let` construction. Notice that we have written the definition of `velocityCF` using the "three-input thinking" mentioned earlier. Exercise 14.1 asks you to rewrite the function using four-input thinking.

We can write a function `positionCF` that produces a position function given mass, initial position, initial velocity, and a list of constant forces.

```
positionCF :: Mass
           -> Position          -- initial position
           -> Velocity          -- initial velocity
           -> [Force]           -- list of forces
           -> Time -> Position  -- position function
positionCF m x0 v0 fs
    = let fNet = sum fs
          a0   = fNet / m
          x t  = x0 + v0 * t + a0*t**2 / 2
      in x
```

Here, we have used Equation 4.15 or 14.5. Returning to Example 14.1, the velocity of the car as a function of time is

```
velocityCF 0.1 0.6 [0.04, -0.08]
```

because 0.1 kg is the mass of the car, 0.6 m/s is its initial velocity, and the square-bracketed list contains the forces in Newtons. We can ask for the type of this function in GHCi, and we can ask for values of the velocity at specific times.

```
Prelude> :l Newton2
[1 of 1] Compiling Newton2          ( Newton2.hs, interpreted )
Ok, one module loaded.
*Newton2> :t velocityCF 0.1 0.6 [0.04, -0.08]
velocityCF 0.1 0.6 [0.04, -0.08] :: Time -> Velocity
*Newton2> velocityCF 0.1 0.6 [0.04, -0.08] 0
```

```
0.6
*Newton2> velocityCF 0.1 0.6 [0.04, -0.08] 1
0.2
```

Since we have the velocity function in hand, we can graph it. Let's write the code to do so first. Most of the code below is for setting up a title, axis labels, and the name of the file we want produced. The interesting stuff is at the end, where we give a list of times at which to evaluate the function and the function itself.

```
carGraph :: IO ()
carGraph
    = plotFunc [Title "Car on an air track"
               ,XLabel "Time (s)"
               ,YLabel "Velocity of Car (m/s)"
               ,PNG "CarVelocity.png"
               ,Key Nothing
               ] [0..4 :: Time] (velocityCF 0.1 0.6 [0.04, -0.08])
```

This code produces the graph in Figure 14-2.

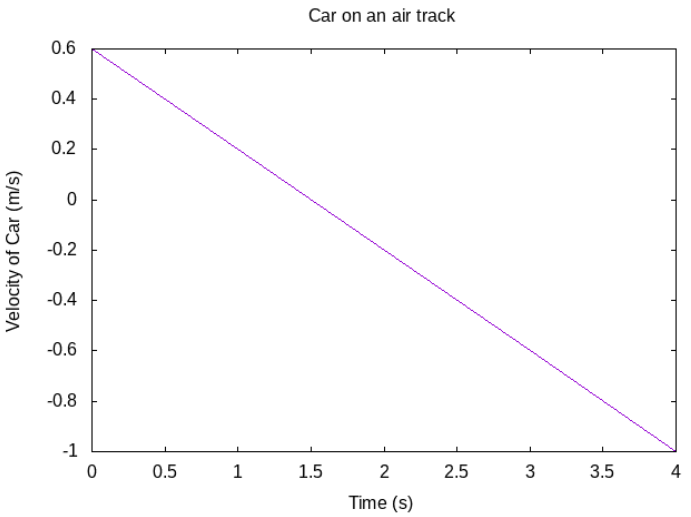

Figure 14-2: Car velocity as a function of time in Example 14.1

If you load this chapter's module, `Newton2`, into GHCi and enter `carGraph`,

```
*Newton2> carGraph
```

you will not get any return value, but the function will produce a Portable Network Graphics (PNG) file named *CarVelocity.png* on your hard drive. Without the `PNG "CarVelocity.png"` option, the `carGraph` function would produce a graph on the screen.

Note that the negative acceleration in the graph in Figure 14-2 (which exists over the entire time interval from $t = 0$ to $t = 4$ s) does not mean that

the car is always slowing down. Rather, a negative acceleration means an acceleration to the west. The car slows down during the first 1.5 s as it is moving east but then begins to speed up as it moves west. When the acceleration and velocity of an object point in the same direction, the object speeds up. When the acceleration and velocity of an object point in opposite directions, the object slows down.

With the functions `velocityCF` and `positionCF`, we have general-purpose ways of solving any Newton's second law type problem in one spatial dimension with constant forces. Next we'll consider forces that change in time.

Second Law with Forces that Depend Only on Time

The next situation for Newton's second law is when the net force depends on time but not on position or velocity. Figure 14-3 shows a schematic diagram for Newton's second law with forces that depend only on time.

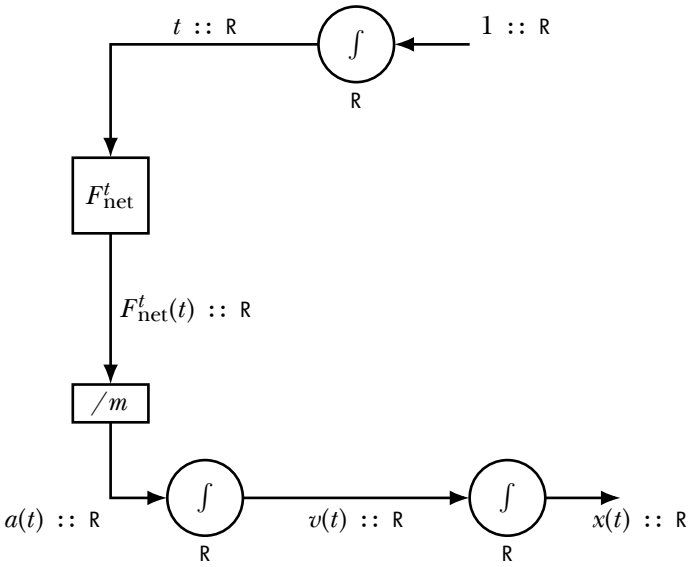

Figure 14-3: Schematic diagram for Newton's second law with forces that depend only on time

The constant number 1 is fed into an integrator to produce a value for time. (The time changes at a rate of 1 second per second.) As usual, wires are labeled with names and types. Integrators are labeled with the type of state they hold. Time is fed into the net force function F^t_{net} (superscript t for time-dependent), which produces net force as output. To obtain the acceleration of the object, we need to divide the net force acting on the object by the object's mass.

$$a(t) = \frac{F^t_{\text{net}}(t)}{m} \tag{14.6}$$

We then integrate the acceleration to obtain the velocity,

$$v(t) = v(0) + \int_0^t a(t')\,dt' = v(0) + \frac{1}{m}\int_0^t F^t_{\text{net}}(t')\,dt' \tag{14.7}$$

and we integrate the velocity to obtain the position:

$$\begin{aligned} x(t) &= x(0) + \int_0^t v(t'')\,dt'' \\ &= x(0) + \int_0^t \left[v(0) + \frac{1}{m}\int_0^{t''} F^t_{\text{net}}(t')\,dt' \right] dt'' \\ &= x(0) + v(0)t + \frac{1}{m}\int_0^t \left[\int_0^{t''} F^t_{\text{net}}(t')\,dt' \right] dt'' \end{aligned}$$

The wires of the diagram represent quantities that are continuously changing in time. Rectangular boxes represent pure functions, whereas circular elements contain state.

To solve Newton's second law problems with forces that depend on time, we'd like a higher-order function that produces a velocity function, similar to `velocityCF` in the previous section. One difference is that now we need to provide a list of force *functions* rather than a list of numerical forces. We want a function `velocityFt` (the `Ft` suffix denotes that forces depend only on time) with the following type signature:

```
velocityFt :: Mass -> Velocity -> [Time -> Force] -> Time -> Velocity
```

Given the mass of our object, its initial velocity, and a list of force functions, we want to produce a velocity function.

Because we're going to do numerical integration to get the velocity function, we'll add one additional parameter to this type signature, namely the time step for numerical integration. Thus, we arrive at the following definition for `velocityFt`:

```
velocityFt :: R                 -- dt for integral
           -> Mass
           -> Velocity          -- initial velocity
           -> [Time -> Force]   -- list of force functions
           -> Time -> Velocity  -- velocity function
velocityFt dt m v0 fs
    = let fNet t = sum [f t | f <- fs]
          a t = fNet t / m
      in antiDerivative dt v0 a
```

In this definition, we begin by naming the inputs: `dt` for an integration time step, `m` for the mass of the object we are attending to, `v0` for the initial velocity of this object, and `fs` for a list of force functions. Note that the local variable for forces, `fs`, had type `[Force]` (or `[R]`) when used in `velocityCF` and `positionCF` for situations with constant forces, but it now has type `[Time -> Force]` (or `[R -> R]`) for situations with forces that depend on time.

We again use a `let` construction to define local functions, a net force function, and an acceleration function. The net force function adds together the forces provided in the list `fs`. We might have hoped we could use the same line of code we used in `velocityCF`, namely `fNet = sum fs`, to sum the forces. After all, `fs` is still a list. The trouble is that `sum` works only with types that are instances of `Num`, as you can see if you look at the type of `sum`. So while it is happy to add numbers (type `R`), it is not happy to add functions (type `R -> R`). Fortunately, we can evaluate the force functions at a time `t` introduced as an argument to `fNet` and then add the resulting numbers.

The acceleration function comes from Newton's second law. Here, we might have hoped that we could divide the net force function by the mass to obtain the acceleration function, perhaps writing `a = fNet / m`. But recall that the division operator insists that it work with two values that have the same type and that this type be an instance of `Fractional`. The division operator does not want to work with functions. Again, we address this by evaluating the `fNet` function at the time `t` introduced as the argument to the acceleration function `a`.

Finally, the velocity comes from taking an antiderivative of the acceleration function. We defined the functions `antiDerivative` and `integral` in Chapter 6, but we'll repeat their definitions here:

```
antiDerivative :: R -> R -> (R -> R) -> (R -> R)
antiDerivative dt v0 a t = v0 + integral dt a 0 t

integral :: R -> (R -> R) -> R -> R -> R
integral dt f a b
    = sum [f t * dt | t <- [a+dt/2, a+3*dt/2 .. b - dt/2]]
```

Note that `velocityFt dt m v0 fs` has type `R -> R` and is the velocity function for an object with mass `m`, initial velocity `v0`, and list of force functions `fs`. This velocity function is part of the solution to the mechanics problem. Another part of the solution is a position function. We can write a function `positionFt` that produces a position function given mass, initial position, initial velocity, and a list of force functions.

```
positionFt :: R                 -- dt for integral
           -> Mass
           -> Position          -- initial position
           -> Velocity          -- initial velocity
           -> [Time -> Force]   -- list of force functions
           -> Time -> Position  -- position function
positionFt dt m x0 v0 fs
    = antiDerivative dt x0 (velocityFt dt m v0 fs)
```

This function works by taking an antiderivative of the velocity function, which we find using `velocityFt`.

As an example of solving Newton's second law with a time-dependent force, consider a child riding a bike. By working the pedals, the child arranges for the ground to apply a constant forward force of 10 N on the bike

for 10 seconds, after which the child coasts for the next 10 seconds. Following the coasting, the child returns to the 10-N force for another 10 seconds, and so on, as illustrated in Figure 14-4.

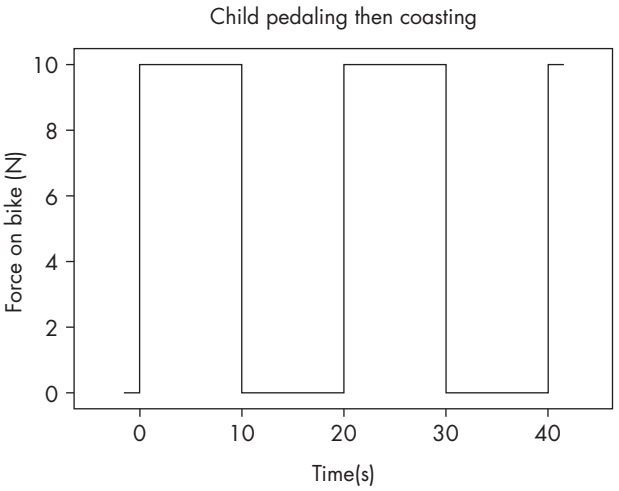

Figure 14-4: Force as a function of time for a child on a bike

In this example, we'll assume that air resistance is not important and that there is only one force on the bike.

Here is the equation for the time-dependent force of pedaling and coasting:

$$F_{\text{pc}}(t) = \begin{cases} 10 \text{ N}, & (20 \text{ s})n \leq t < (20 \text{ s})n + 10 \text{ s for some integer } n \\ 0 \text{ N}, & (20 \text{ s})n + 10 \text{ s} \leq t < (20 \text{ s})n + 20 \text{ s for some integer } n \end{cases} \quad (14.8)$$

The force is either 0 N or 10 N, depending on where the time falls in a 20-second cycle. If the time falls in the first 10 seconds of the cycle, the force is 10 N. If, on the other hand, the time falls in the last 10 seconds of the cycle, the force is 0 N.

Here is the time-dependent force of Equation 14.8 in Haskell:

```
pedalCoast :: Time -> Force
pedalCoast t
    = let tCycle = 20
          nComplete :: Int
          nComplete = truncate (t / tCycle)
          remainder = t - fromIntegral nComplete * tCycle
      in if remainder < 10
         then 10
         else 0
```

The local variable `tCycle` is the number of seconds for a full cycle. The variable `nComplete` uses the Prelude function `truncate` to calculate the number of complete cycles from the time `t`. The `truncate` function produces a type

with type class `Integral` (recall `Integer` and `Int` are instances of `Integral`). We provide a local type signature to say that we want `nComplete` to have type `Int`. The local type signature is optional, but the compiler will give us a warning that it chose a default type if we don't specify something. Remove the local type signature to see what the warning looks like. This is a mild warning. We don't mind that the compiler chooses `Integer` instead of `Int`. You can feel free to ignore this warning and use the code without the type signature if you wish.

The `remainder` is the number of seconds, between 0 and 20, that have elapsed since the beginning of the most recent cycle. We want `remainder` to be a real number, so we must use `fromIntegral` to convert `nComplete :: Int` into a real number.

Figure 14-5 shows the position of the child as a function of time.

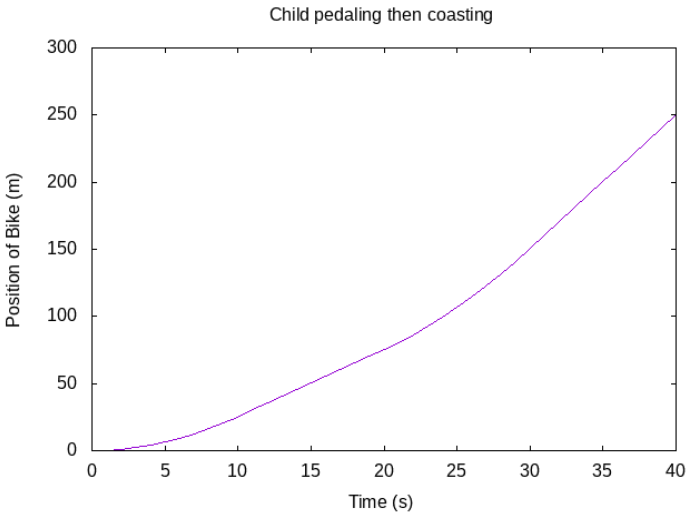

Figure 14-5: Position as a function of time for the child on a bike

Here is the Haskell code that produced Figure 14-5:

```
childGraph :: IO ()
childGraph
    = plotFunc [Title "Child pedaling then coasting"
               ,XLabel "Time (s)"
               ,YLabel "Position of Bike (m)"
               ,PNG "ChildPosition.png"
               ,Key Nothing
               ] [0..40 :: R] (positionFt 0.1 20 0 0 [pedalCoast])
```

The most interesting part of the code is the last line, where we specify the function we want plotted. This function, `positionFt 0.1 20 0 0 [pedalCoast]`, uses the `positionFt` function we developed earlier in the chapter with a time step of 0.1 s, a mass of 20 kg, 0s for initial position and initial velocity, and a list of forces that includes only the force of pedaling and coasting. All of the

relevant physical information is included in the "name" of the function we are plotting.

You can see from the graph in Figure 14-5 that during the first 10 seconds, the child's position curve is parabolic, as we'd expect from constant acceleration. From 10 to 20 seconds, the position shows constant velocity while the child is coasting. From 20 to 30 seconds, there is another period of acceleration in which the position curve is parabolic, followed by a second period of coasting.

With the functions `velocityFt` and `positionFt`, we have general-purpose ways of solving any Newton's second law type problem in one spatial dimension with forces that depend only on time. We're now ready to look at forces that depend on velocity, the most common of which is air resistance.

Air Resistance

In this section, we'll make a short diversion from our path of considering Newton's second law in the presence of forces that depend on time, velocity, neither, or both to develop an expression for the force of air resistance on an object in one-dimensional motion. Air resistance is a force that depends only on velocity, and we'll use it in the next several sections as we develop ways to solve Newton's second law with forces that depend on velocity.

Introductory physics classes typically ignore air resistance or treat it very lightly, because the presence of air resistance turns Newton's second law into a differential equation, which is considered beyond the scope of an introductory physics course. In this chapter and the next, we'll develop numerical methods for solving differential equations, meaning that air resistance is not something we want to avoid; in fact, it showcases the power of our tools.

To develop a model of air resistance, let's think of the interaction between an object and the air around it as a collision. Suppose the object is moving with velocity v. In this section, v represents the real-valued, one-dimensional velocity of the object (a quantity with type `R`) and not the velocity function or the speed.

Let the cross-sectional area of the object be A and the density of air be ρ. We analyze the motion of the object over a small time interval dt. We assume that the initial velocity of the air is 0, and that the final velocity of the air is v (in other words, after the collision, the air is traveling at the same speed as the object).

The distance the object travels in time dt is $v\,dt$. The volume of air swept out by the object in time dt is $Av\,dt$. The mass of air disturbed by the object in time dt is $\rho Av\,dt$. The momentum imparted to the air by the object in time dt is the product of the mass of the air, $\rho Av\,dt$, and the change in velocity of the air, which is v, as we assume that the air starts from rest and ends the short time interval with velocity v. The momentum imparted to the air is $\rho Av^2\,dt$. The force felt by the air is this change in momentum per unit time, or ρAv^2. The force felt by the object from the air is equal and opposite to this following Newton's third law, which we will discuss in Chapter 19.

Our derivation was really quite approximate because we don't know that the air molecules really end up with velocity v, and we haven't even tried to account for the forces of air molecules on each other as the air compresses. Nevertheless, the form of our result is quite useful and approximately correct. Objects with different shapes respond a bit differently though, so it is useful to introduce a *drag coefficient* C to account for these differences. The drag coefficient is a dimensionless constant that is a property of the object that is flying through the air. It is also conventional to include a factor of $1/2$ so that the magnitude of the force of air resistance on the object is $C\rho A v^2/2$. This expression is never negative. We would prefer an expression in which the force is negative when the velocity is positive and positive when the velocity is negative. Our final expression for the one-dimensional force of air resistance is

$$F_{\text{air}}(v) = -\frac{1}{2}C\rho A\,|v|\,v \tag{14.9}$$

where the minus sign and the absolute value ensure that the force acts in a direction opposite the velocity. Air resistance is acting to slow the object. In Haskell, we'll write Equation 14.9 for air resistance as follows:

```
fAir :: R  -- drag coefficient
     -> R  -- air density
     -> R  -- cross-sectional area of object
     -> Velocity
     -> Force
fAir drag rho area v = -drag * rho * area * abs v * v / 2
```

In the mathematical notation of Equation 14.9, we're treating F_{air} as a function of one variable. The parameters C, ρ, and A are not listed explicitly as variables that F_{air} depends on. Eliding parameters like this is standard practice in physics, but in some sense it's an abuse of notation. In the Haskell notation, we must include all of the variables that the force of air resistance depends on. We list the three parameters first, before the `Velocity`, so that an expression like `fAir 1 1.225 0.6` is a fully legitimate function that takes only velocity as input. The function `fAir 1 1.225 0.6` has already chosen `drag = 1`, `rho = 1.225`, and `area = 0.6`.

With this brief foray into air resistance, and particularly the development of Equation 14.9, we're now ready to look at Newton's second law in the case where forces on our object depend only on its velocity.

Second Law with Forces that Depend Only on Velocity

The next situation for Newton's second law is when the net force depends on velocity but not on time or position. What we really mean here is that the forces do not depend *explicitly* on time. Velocity is a function that depends on time, and forces are allowed to depend on the velocity in this section, so there is a sense in which the forces depend on time. The constraint in this section is that the forces can depend on time *only through the velocity*.

The force functions may depend only on one variable, the velocity. We use F_j^v to denote the jth force function of one variable that gives force when supplied with velocity and we use F_{net}^v to denote the function of one variable that gives net force when supplied with velocity.

$$F_{\text{net}}^v(v_0) = \sum_j F_j^v(v_0)$$

We use v_0 as a local variable for velocity (type `R`) rather than v in this section because we want v to stand for the velocity function of our object (type `R -> R`).

Figure 14-6 shows a schematic diagram for Newton's second law with forces that depend only on velocity.

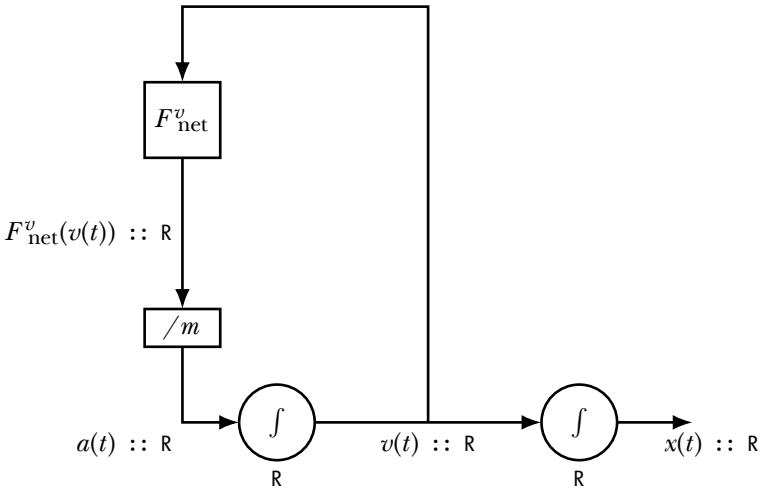

Figure 14-6: Newton's second law with forces that depend only on velocity

This diagram, unlike previous diagrams, contains a loop. The velocity produced by the integrator of acceleration serves as the input to the net force function. The loop in the diagram indicates that Newton's second law produces a differential equation. Because the loop contains one integrator, we get a first-order differential equation. A differential equation is a more difficult mathematical problem than a mere integral or antiderivative, as we had when forces depended only on time.

Newton's second law is given by the following equation:

$$\frac{d}{dt}\left[v(t)\right] = \frac{1}{m}\sum_j F_j^v(v(t)) \tag{14.10}$$

The information this equation represents is the same as the information in the schematic diagram of Figure 14-6. The equation describes how the rate of change of velocity depends on velocity itself through the forces that act on the object. The function `newtonSecondV`, presented next, is yet a third way to express Newton's second law; this function returns the rate of change

of velocity when given the current value of velocity along with the forces that act on the object.

```
newtonSecondV :: Mass
              -> [Velocity -> Force]  -- list of force functions
              -> Velocity             -- current velocity
              -> R                    -- derivative of velocity
newtonSecondV m fs v0 = sum [f v0 | f <- fs] / m
```

We can integrate the acceleration to obtain the velocity.

$$v(t) = v(0) + \int_0^t a(t')\,dt' = v(0) + \frac{1}{m}\int_0^t F^v_{\text{net}}(v(t'))\,dt'$$

Unlike the case with time-dependent forces, we cannot simply perform the integral here because the velocity function we are trying to find appears under the integral. How to proceed?

To solve the differential equation, Equation 14.10, we will discretize time, which is something we have been doing with our numerical derivatives and integrals when we chose a time step. As long as our time step Δt is smaller than any important time scales in the situation we are addressing, the slope of the line connecting points $(t, v(t))$ and $(t + \Delta t, v(t + \Delta t))$ will be approximately equal to the derivative of velocity at time t.

$$\frac{v(t + \Delta t) - v(t)}{\Delta t} \approx \frac{dv(t)}{dt}$$

Rearranging this equation leads to the *Euler method* for solving a first-order differential equation.

$$v(t + \Delta t) \approx v(t) + \frac{dv(t)}{dt}\Delta t \tag{14.11}$$

The Euler method approximates the velocity at $t + \Delta t$ by the sum of the velocity at t and the product of the derivative at t with the time step Δt. The Euler method gives a way to find velocity at a later time from velocity at an earlier time if we know the derivative of velocity at the earlier time.

Figure 14-7 pictorially describes the Euler method for solving Newton's second law.

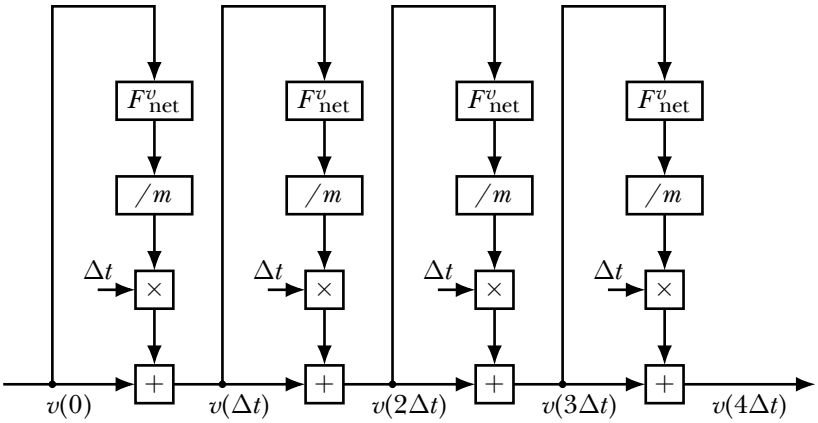

Figure 14-7: Euler method for Newton's second law in one dimension, for the special case in which net force depends only on velocity

The diagram shows how data is acted on by pure functions to compute the velocity of the object at different times. Because the diagram employs only pure functions (functions that do nothing but return an output from inputs and unchanging global values), we refer to this as a *functional diagram*. Whereas the schematic diagram in Figure 14-6 presents time as continuous, this diagram shows time as discrete. And whereas the schematic diagram has wires with values that are continuously changing in time, the functional diagram has wires with values that do not change. Different points in time have different wires in the functional diagram. While a schematic diagram may contain the stateful integrator from Figure 6-5, a functional diagram uncoils and replaces the integrator with a discrete, functional model like the one in Figure 6-7. We can see from Figure 14-7 that the same set of computations occurs at each time step to produce a new velocity from an old velocity. We call the set of computations that occurs at each time step the *velocity-update function*.

Figure 14-8 shows the velocity-update function, which is based on the application of the Euler method to one small time step.

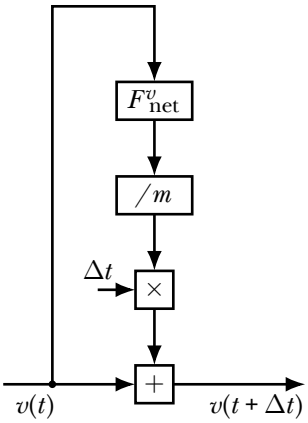

Figure 14-8: Velocity-update function used in the Euler method for solving Newton's second law with forces that depend only on velocity

Figure 14-8 shows a functional diagram for velocity update, visually describing how velocity at $t + \Delta t$ is computed from velocity at t and the forces.

Here is the velocity-update equation showing how a new velocity is obtained from an old velocity:

$$v(t + \Delta t) = v(t) + \frac{F^v_{\text{net}}(v(t))}{m}\Delta t \tag{14.12}$$

Lastly, we have the Haskell function `updateVelocity`, which advances the value of the velocity by one time step.

```
updateVelocity :: R                    -- time interval dt
               -> Mass
               -> [Velocity -> Force]  -- list of force functions
               -> Velocity             -- current velocity
               -> Velocity             -- new velocity
updateVelocity dt m fs v0
    = v0 + (newtonSecondV m fs v0) * dt
```

The functional diagram in Figure 14-8, the velocity-update equation (Equation 14.12), and the function `updateVelocity` express the same information in different forms, namely how to take one step in time with the Euler method.

Now we want to write a function `velocityFv`, similar to `velocityCF` and `velocityFt`, but for the case of forces that depend on velocity. To think of `updateVelocity` as a function that takes `Velocity` as input and gives `Velocity` as output, we want to think of the time step, mass, and list of force functions as parameters. The function `updateVelocity dt m fs` has type `Velocity -> Velocity` and plays the role of the iterable function `f` in Figure 6-4 on page 76.

```
velocityFv :: R                    -- time step
           -> Mass
```

```
          -> Velocity             -- initial velocity v(0)
          -> [Velocity -> Force]  -- list of force functions
          -> Time -> Velocity     -- velocity function
velocityFv dt m v0 fs t
    = let numSteps = abs $ round (t / dt)
      in iterate (updateVelocity dt m fs) v0 !! numSteps
```

We define a local variable `numSteps` to be the number of time steps we need to take to get as close as possible to the desired time `t`. We iterate the function `updateVelocity dt m fs`, starting at the initial velocity `v0`, and then select the single value of velocity from this infinite list that is closest to the desired time.

As an example of a situation with forces that depend only on velocity, let's consider a bicycle rider heading north on a flat, level road. We'll consider two forces in this situation. First, there is the northward force that the road exerts on the tires of the bicycle because the rider is working the pedals. Let us call this force F_{rider} (it is directly produced by the road on the bike, but it is indirectly produced by the rider), and assume that this force is a constant 100 N. Second, there is the southward force of air resistance that impedes the northward progress of the rider, especially when she is traveling quickly. We'll use the expression for air resistance that we developed in the previous section with Equation 14.9. The net force is

$$\begin{aligned} F^v_{\text{net}}(v_0) &= F_{\text{rider}} + F_{\text{air}}(v_0) \\ &= F_{\text{rider}} - \frac{1}{2} C\rho A \left|v_0\right| v_0 \end{aligned}$$

Let's take the mass of the bike plus rider to be $m = 70$ kg. We'll choose a drag coefficient of $C = 2$, take the density of air to be $\rho = 1.225$ kg/m^3, and approximate the cross-sectional area of bike and rider to be 0.6 m^2. Starting from rest, our mission is to find the velocity of the bike as a function of time.

Before we use our Haskell functions to investigate the motion of the bike, we're going to show how to use the Euler method by hand.

Euler Method by Hand

Let's use the Euler method by hand to compute the first several values of velocity for the bike. The purpose in doing this is to get a clear understanding of what is happening in the Euler method, so the code we write will be meaningful and not just a formal representation of some abstract vague process. We choose a time step of 0.5 s. Our mission is to complete the following table. We can fill in all of the time values because they are simply spaced at 0.5 s intervals. The initial velocity is 0, so we fill that in as well.

t (s)	v(t) (m/s)
0.0	0.0000
0.5	
1.0	
1.5	

We will complete the table by using Equation 14.12 to update the velocity over and over again. To compute the velocity at 0.5 s, we choose $t = 0$ in Equation 14.12.

$$\begin{aligned} v(0.5\text{ s}) &= v(0.0\text{ s}) + \frac{F^v_{\text{net}}(v(0.0\text{ s}))(0.5\text{ s})}{70\text{ kg}} \\ &= 0.0000\text{ m/s} + \frac{F^v_{\text{net}}(0.0000\text{ m/s})(0.5\text{ s})}{70\text{ kg}} \\ &= 0.0000\text{ m/s} + \frac{(100\text{ N})(0.5\text{ s})}{70\text{ kg}} \\ &= 0.7143\text{ m/s} \end{aligned}$$

We update the table with

t (s)	v(t) (m/s)
0.0	0.0000
0.5	0.7143
1.0	
1.5	

and then we calculate $v(1.0\text{ s})$ using Equation 14.12 with $t = 0.5$ s:

$$\begin{aligned} v(1.0\text{ s}) &= v(0.5\text{ s}) + \frac{F^v_{\text{net}}(v(0.5\text{ s}))(0.5\text{ s})}{70\text{ kg}} \\ &= 0.7413\text{ m/s} + \frac{F^v_{\text{net}}(0.7143\text{ m/s})(0.5\text{ s})}{70\text{ kg}} \\ &= 0.7413\text{ m/s} \\ &\quad + \frac{[100\text{ N} - (1)(1.225\text{ kg/m}^3)(0.6\text{ m}^2)(0.7143\text{ m/s})^2](0.5\text{ s})}{70\text{ kg}} \\ &= 1.4259\text{ m/s} \end{aligned}$$

We add this to the appropriate row of the table and continue.

$$\begin{aligned} v(1.5\text{ s}) &= v(1.0\text{ s}) + \frac{F^v_{\text{net}}(v(1.0\text{ s}))(0.5\text{ s})}{70\text{ kg}} \\ &= 1.4259\text{ m/s} + \frac{F^v_{\text{net}}(1.4259\text{ m/s})(0.5\text{ s})}{70\text{ kg}} \\ &= 1.4259\text{ m/s} \\ &\quad + \frac{[100\text{ N} - (1)(1.225\text{ kg/m}^3)(0.6\text{ m}^2)(1.4259\text{ m/s})^2](0.5\text{ s})}{70\text{ kg}} \\ &= 2.1295\text{ m/s} \end{aligned}$$

The completed table looks like this:

t (s)	v(t) (m/s)
0.0	0.0000
0.5	0.7143
1.0	1.4259
1.5	2.1295

Euler Method in Haskell

Now we'll use the `velocityFv` function to calculate velocity for the bike. Here is a velocity function for the bike with a time step of 1 s:

```
bikeVelocity :: Time -> Velocity
bikeVelocity = velocityFv 1 70 0 [const 100,fAir 2 1.225 0.6]
```

The higher-order function `const` can be used to make a constant function. The function `const 100` takes one input, ignores it, and returns 100 as output. It is equivalent to the anonymous function `\_ -> 100`. We're using it here to represent the constant force of 100 N.

Notice the data that must be supplied to solve the bike problem. We provide the 70-kg mass, the 0 m/s initial velocity of the bike, and the two forces: `const 100`, a constant force of 100 N, and `fAir 2 1.225 0.6`, which is the force of air resistance with a drag coefficient of 2, an air density of 1.225 kg/m^3, and a cross-sectional area of 0.6 m^2.

Here is the code to produce a graph of velocity versus time:

```
bikeGraph :: IO ()
bikeGraph = plotFunc [Title "Bike velocity"
                     ,XLabel "Time (s)"
                     ,YLabel "Velocity of Bike (m/s)"
                     ,PNG "BikeVelocity1.png"
                     ,Key Nothing
                     ] [0,0.5..60] bikeVelocity
```

The code plots the `bikeVelocity` function, including a title and axis labels, and makes a PNG file that can be included in another document. Figure 14-9 contains the graph itself.

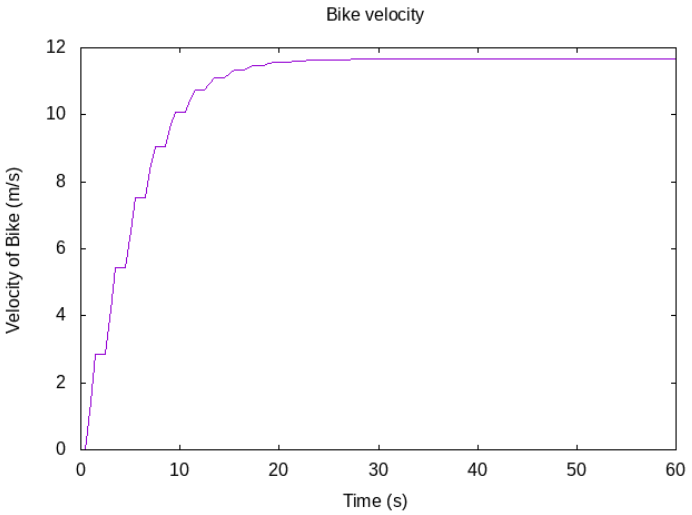

Figure 14-9: Bike velocity as a function of time. The stair-stepping look can be fixed and is discussed in the text.

A phenomenon occurs in Figure 14-9 that does not occur in constant acceleration situations: the establishment of a terminal velocity. After 20 seconds or so, the forward force of the road (from the pedaling) matches the backward force of the air. At this point we have no net force (or a very small net force), and the velocity stays at the terminal velocity.

Why the stair-stepping look to Figure 14-9? We used a time step of one second to do the calculation of the velocity function `bikeVelocity`, but then we asked the `plotFunc` function to give us a plot of that function every half a second. If we want a smooth plot, we have a couple of options. The simplest would be to ask for a plot with time values spaced at least one second apart. Alternatively, we could calculate the `bikeVelocity` function using a smaller time step. In any case, we shouldn't ask for more resolution in the graph than we asked for in the function we are graphing.

With the functions `velocityFv` and `positionFv`, the latter of which you are asked to write in Exercise 14.4, we have general-purpose tools for solving any Newton's second law type problem in one spatial dimension with forces that depend only on velocity. Before we turn to the case in which forces depend on both time and velocity, let's take a moment to view what we've just done from a broader perspective.

The State of a Physical System

A fruitful way to structure our thinking about Newton's second law, and also later about the Maxwell equations, revolves around the concept of the *state* of a physical system, which is the collection of information needed to say precisely what is going on with the system *at a particular instant of time*.

The state represents the current "state of affairs" of the system, containing enough information that future prediction can be based on the current state instead of past information about the system. The state evolves in time, changing according to some rule.

Given a physical system that we wish to understand, the state-based paradigm suggests the following conceptual division:

1. What information is required to specify the state of the system?
2. What is the state at some initial time?
3. By what rule does the state change with time?

When we treated Newton's second law with constant forces and forces that depend only on time, we did not use a state-based method because we did not need one. In those cases, we could use algebra or integration to find how the position and velocity of our object changed in time. When we looked at forces that depend on velocity, we had a schematic diagram with a loop that corresponded to a differential equation, shown in Figure 14-6. The state-based method is particularly useful for differential equations.

There are three things to notice about Figure 14-6 that relate to the state-based method. First, notice that there is one integrator in the loop and that this integrator holds the value of velocity as state. Second, notice that the differential equation, Equation 14.10, gives an expression for the rate of change of velocity. Lastly, notice that the forces depend on velocity. For these three reasons, in the case where forces depend only on velocity, the state of the object consists of the velocity of the object.

In general, the answer to question 1 is a data type. The state of an object experiencing forces that depend only on the object's velocity is a value of the data type `Velocity`. In the next section, where forces depend on time and velocity, the data type we will use for state is the pair `(Time,Velocity)`. As we consider more complex physical situations, the data type we use to hold the state of our physical system will contain more information.

Question 2 above is, in some sense, the smallest question. It may even be possible to do some analysis without an answer to question 2. But if we wish to know properties of a system at a later time, then we wish to know the state at a later time, and this typically requires knowing the state at some earlier time. The answer to question 2 is a value of the data type from question 1.

Question 3 requires a physical theory to answer. In the case of mechanics, Newton's second law gives the rule by which the state changes in time.

Let's see how the state-based method applies in the case where the forces on an object depend only on time and the velocity of the object.

Second Law with Forces that Depend on Time and Velocity

The next situation for Newton's second law is when the forces depend on both time and velocity but not on position. The force functions depend on

two variables, time and velocity. We use F^{tv}_j to denote the jth function of two variables that gives a force when supplied with time and velocity, and we use F^{tv}_{net} to denote the function of two variables that gives net force when supplied with time and velocity.

$$F^{tv}_{\text{net}}(t, v_0) = \sum_j F^{tv}_j\,(t, v_0)$$

Figure 14-10 shows a schematic diagram for Newton's second law with forces that depend on time and velocity.

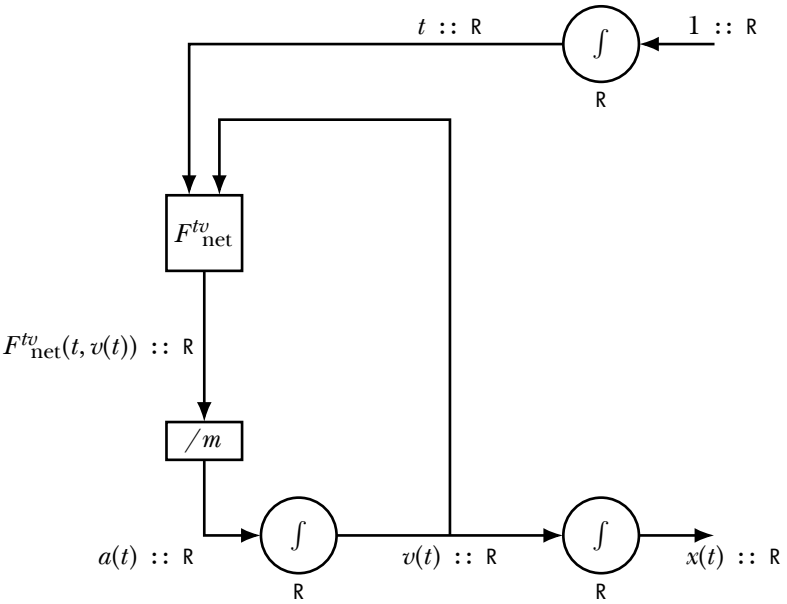

Figure 14-10: Newton's second law with forces that depend on time and velocity

The schematic diagram contains a loop, so Newton's second law is a differential equation, given in Equation 14.14.

$$\frac{d}{dt}\left[t\right] = 1 \tag{14.13}$$

$$\frac{d}{dt}\left[v(t)\right] = \frac{1}{m}\sum_j F^{tv}_j\,(t, v(t)) \tag{14.14}$$

Notice that there is one integrator in the loop in Figure 14-10, which holds the value of velocity as state. There is a way to solve this differential equation using only velocity as the state of the object. However, since the rate of change of velocity in Equation 14.14 depends on both time and velocity (because the forces depend on time and velocity), the state-based method

is simpler to apply if we allow both time and velocity to be *state variables*. This is to say that the data type we will use for state is `(Time,Velocity)`. The difference between Equation 14.10, which expresses Newton's second law with forces that depend only on velocity, and Equation 14.14, which expresses Newton's second law with forces that depend on time and/or velocity, is simply that we need to know the current value of time in the latter case but not in the former. Including time in the state `(Time,Velocity)` is a simple way to gain convenient access to the current time.

Which quantities deserve to be called state variables? Say I have a particle in space acted on by a known (time-independent) force law. The state variables are the position and velocity because we can calculate the position and velocity at the next time instant from them. Why is acceleration not a state variable? To use the terminology of earlier sections in this chapter, state variables are numbers that identify a particular solution to the differential equation—they are the initial values that convert integrals into antiderivatives. Time is usually not considered a state variable, but taking it as one makes it easier to think about time-dependent forces. Readers interested in a more in-depth discussion of state variables and their uses are encouraged to see [16] and [17].

The Haskell function `newtonSecondTV`, shown below, expresses Newton's second law in the case where forces depend on time and velocity.

```
newtonSecondTV :: Mass
               -> [(Time,Velocity) -> Force]  -- force funcs
               -> (Time,Velocity)             -- current state
               -> (R,R)                       -- deriv of state
newtonSecondTV m fs (t,v0)
    = let fNet = sum [f (t,v0) | f <- fs]
          acc = fNet / m
      in (1,acc)
```

Given the mass of an object and a list of forces that act on the object, now expressed as functions of the state `(Time,Velocity)`, `newtonSecondTV` gives instructions for computing the time derivatives of the state variables from the state variables themselves. The return type `(R,R)` is meant to stand for time derivative of time, which is always the dimensionless number 1, and time derivative of velocity, which is acceleration. The acceleration is computed from Newton's second law by finding the net force and dividing by the mass.

To solve Equation 14.14, we will discretize time and use the Euler method. We'll continue to use Equation 14.11 for the Euler method. Figure 14-11 pictorially describes the Euler method for solving Newton's second law when forces depend on time and/or velocity.

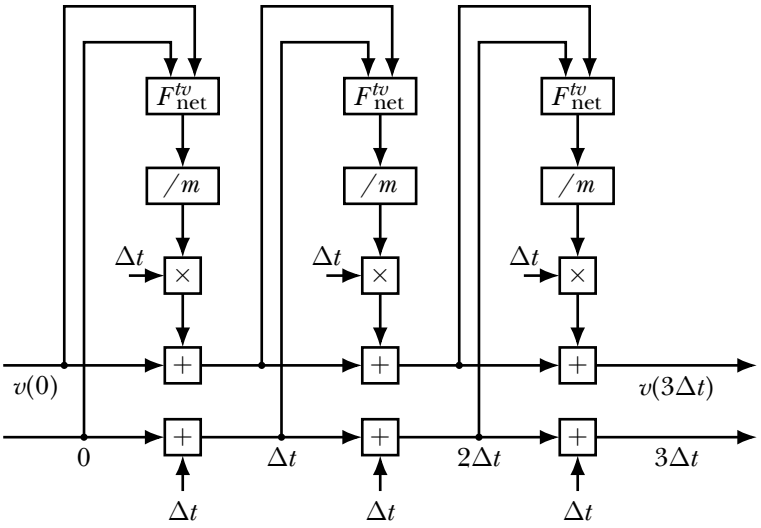

Figure 14-11: Euler method for Newton's second law in one dimension, for the special case in which net force depends only on time and/or velocity

The diagram shows how functions act on the state variables at one point in time to compute the state variables at the next point in time. The same set of computations reoccurs at each time step to produce a new state from an old state. We call the set of computations that occurs at each time step the *state-update function*.

The state-update function is shown pictorially in Figure 14-12. The figure shows a functional diagram for state update, visually describing how time and velocity at $t + \Delta t$ are computed from time and velocity at t, given the force functions.

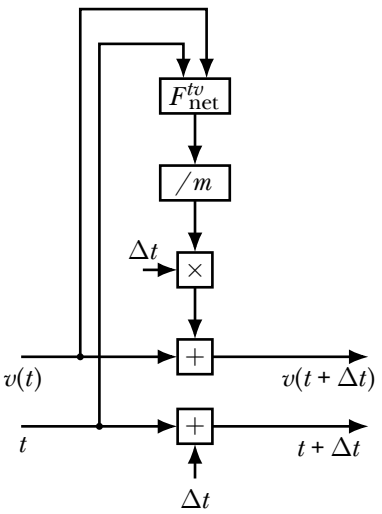

Figure 14-12: Euler method update for Newton's second law with forces that depend only on time and velocity

Here are the state-update equations showing how the new state variables are obtained from the old state variables:

$$t + \Delta t = t + 1\Delta t \tag{14.15}$$

$$v(t + \Delta t) = v(t) + \frac{F^{tv}_{\text{net}}(t, v(t))}{m}\Delta t \tag{14.16}$$

Equations 14.15 and 14.16 are state-update equations for an object exposed to forces that depend on time and velocity. The state-update equations tell us how the state variables time and velocity must be updated to advance to the next time step. The time update in Equation 14.15 is easy: we just add Δt to the old time to get the new time. To update the velocity in Equation 14.16, we compute an acceleration, multiply by a time step to get a change in velocity, and add that change to the old velocity. Applying these state-update equations is how we carry out the Euler method for solving a differential equation. This state-update procedure is the main tool we will use to solve problems in Newtonian mechanics.

The following Haskell function `updateTV`, named because it updates both time and velocity, advances the value of the state by one time step.

```
updateTV :: R                           -- time interval dt
         -> Mass
         -> [(Time,Velocity) -> Force]  -- list of force funcs
         -> (Time,Velocity)             -- current state
         -> (Time,Velocity)             -- new state
updateTV dt m fs (t,v0)
    = let (dtdt, dvdt) = newtonSecondTV m fs (t,v0)
      in (t  + dtdt * dt
         ,v0 + dvdt * dt)
```

The function `updateTV` takes a few parameters and produces a function with type `(Time,Velocity) -> (Time,Velocity)`. The third input of `updateTV`, named `fs` with type `[(Time,Velocity) -> Force]`, could have been an input with type `[Time -> Velocity -> Force]`; it's a matter of style, and either choice will work just fine. Here I chose the former, as time and velocity are already paired in the function output.

The time-velocity pair we are passing around in this function represents the state of the object to which we are applying Newton's second law. The function `updateTV` is then an example of a state-update function. In an earlier section, when forces depended only on velocity, the velocity alone acted as state, and the function `updateVelocity` was the appropriate state-update function.

The functional diagram in Figure 14-12, Equations 14.15 and 14.16, and the function `updateTV` express the same information in different forms, namely how to take one step in time with the Euler method.

Depending on what we want to calculate, there are two things we might do with the `updateTV` function, corresponding to two types of representation of the time-velocity data. First, we may wish to produce a list of time-velocity

pairs. Second, we may wish to produce velocity as a function of time. We'll develop functions for these two purposes in the next two subsections.

Method 1: Produce a List of States

A list of time-velocity pairs can be regarded as a solution to a Newton's second law problem with forces that depend on time and velocity because a time-velocity pair gives the state. The list of states contains a time-velocity pair for each time that has been probed by the Euler method in Figure 14-11. The function `statesTV` produces a list of time-velocity pairs when given a time step, a mass, an initial state, and a list of force functions.

```
statesTV :: R                           -- time step
         -> Mass
         -> (Time,Velocity)             -- initial state
         -> [(Time,Velocity) -> Force]  -- list of force funcs
         -> [(Time,Velocity)]           -- infinite list of states
statesTV dt m tv0 fs
    = iterate (updateTV dt m fs) tv0
```

We use `iterate` to achieve the repeated composition in Figure 14-11. But which function do we want to iterate? It's not simply `updateTV` because `updateTV` takes three parameters as input before the time-velocity pair. The function we iterate must have type `a -> a`, or in this case `(Time,Velocity) -> (Time,Velocity)`. The solution is to give `updateTV` its first three parameters to form the function we send to `iterate`. The function we want to iterate is `updateTV dt m fs`, starting with the initial time-velocity pair `tv0`.

The function `statesTV` gives a general-purpose way of solving any Newton's second law type problem in one spatial dimension with forces that depend only on time and velocity. By a solution, we mean an infinite list of states (time-velocity pairs) of the object, spaced one time step apart from each other.

Method 2: Produce a Velocity Function

Now we want to write a function, `velocityFtv`, that is similar to `velocityCF`, `velocityFt`, and `velocityFv`, but for the case of forces that depend on time and velocity. We'll use the infinite list produced by `statesTV`, picking out the particular time-velocity pair that comes closest to our desired time and using the Prelude function `snd` to return the velocity, unpaired from the time.

```
velocityFtv :: R                           -- time step
            -> Mass
            -> (Time,Velocity)             -- initial state
            -> [(Time,Velocity) -> Force]  -- list of force funcs
            -> Time -> Velocity            -- velocity function
velocityFtv dt m tv0 fs t
    = let numSteps = abs $ round (t / dt)
      in snd $ statesTV dt m tv0 fs !! numSteps
```

With the functions `velocityFtv` and `positionFtv`, the latter of which you will be asked to write in Exercise 14.9, we have general-purpose ways of solving any Newton's second law type problem in one spatial dimension with forces that depend only on time and velocity. Let's now take a look at a situation that involves just such forces.

Example: Pedaling and Coasting with Air Resistance

As an example of a situation with forces that depend on time and velocity, let's reconsider our child bicycle rider who is pedaling and coasting, but now in the presence of air resistance. We'll consider two forces in this situation. First, there's the time-dependent force $F_{\text{pc}}(t)$ of pedaling from Equation 14.8. Second, there's the force of air resistance $F_{\text{air}}(v_0)$ that impedes the motion of the child, for which we'll use Equation 14.9. The net force is

$$F^{tv}_{\text{net}}(t, v_0) = F_{\text{pc}}(t) + F_{\text{air}}(v_0)$$

The mass of the bike plus child is $m = 20$ kg. We'll choose a drag coefficient of $C = 2$, take the density of air to be $\rho = 1.225$ kg/m^3, and approximate the cross-sectional area of bike and rider to be 0.5 m^2. Starting from rest, our mission is to find the velocity of the bike as a function of time.

We update the velocity with Equation 14.16. Before we use our Haskell functions to investigate the motion of the bike, we'll show how to use the Euler method by hand.

Euler Method by Hand

Let's use the Euler method by hand to compute several values of velocity for the bike. Again, the purpose in doing the Euler method by hand is simply to get a clear picture of how the state variables get updated in the Euler method. We'll pick a time step of 6 s, even though this is too big to get accurate results, as it is not small compared to relevant time scales, such as the 20-second cycle time. We choose a time step of 6 s so we can sample both pedaling and coasting over the first few time steps. Our mission is to complete the following table. We can fill in all of the time values because they are simply spaced at six-second intervals. The initial velocity is 0, so we'll fill that in as well.

t (s)	v(t) (m/s)
0	0.0000
6	
12	
18	

The force of pedaling is either 10 N or 0 N, depending on the value of the time.

$$F_{\text{pc}}(0\text{ s}) = F_{\text{pc}}(6\text{ s}) = 10\text{ N}$$
$$F_{\text{pc}}(12\text{ s}) = F_{\text{pc}}(18\text{ s}) = 0\text{ N}$$

Repeatedly applying Equation 14.16, we obtain the following:

$$\begin{aligned}
v(6\text{ s}) &= v(0\text{ s}) + \frac{F_{\text{net}}^{tv}(0\text{ s}, v(0\text{ s}))(6\text{ s})}{20\text{ kg}} \\
&= 0.0000\text{ m/s} + \frac{(10\text{ N})(6\text{ s})}{20\text{ kg}} \\
&= 3.0000\text{ m/s} \\
v(12\text{ s}) &= v(6\text{ s}) + \frac{F_{\text{net}}^{tv}(6\text{ s}, v(6\text{ s}))(6\text{ s})}{20\text{ kg}} \\
&= 3.0000\text{ m/s} + \frac{[(10\text{ N}) - (1)(1.225\text{ kg/m}^3)(0.5\text{ m}^2)(3.0000\text{ m/s})^2](6\text{ s})}{20\text{ kg}} \\
&= 4.3463\text{ m/s} \\
v(18\text{ s}) &= v(12\text{ s}) + \frac{F_{\text{net}}^{tv}(12\text{ s}, v(12\text{ s}))(6\text{ s})}{20\text{ kg}} \\
&= 4.3463\text{ m/s} + \frac{[(0\text{ N}) - (1)(1.225\text{ kg/m}^3)(0.5\text{ m}^2)(4.3463\text{ m/s})^2](6\text{ s})}{20\text{ kg}} \\
&= 0.8752\text{ m/s}
\end{aligned}$$

The completed table looks like this:

t (s)	v(t) (m/s)
0	0.0000
6	3.0000
12	4.3463
18	0.8752

Let's turn now to Haskell, using each of the two methods we discussed earlier.

Method 1: Produce a List of States

Here we'll use the function `statesTV` to produce an infinite list of velocity-time pairs called `pedalCoastAir` for the child on the bike.

```
pedalCoastAir :: [(Time,Velocity)]
pedalCoastAir = statesTV 0.1 20 (0,0)
                [\(t,_) -> pedalCoast t
                ,\(_,v) -> fAir 2 1.225 0.5 v]
```

Notice the data that must be supplied to solve this problem. We provide a 0.1-s time step, the 20-kg mass, an initial state consisting of 0 for the time and 0 for the velocity, and the two forces, expressed here as anonymous functions. The function `pedalCoast` is a function only of time, so it cannot be listed directly as a force function because a force function for `statesTV` takes

a time-velocity pair as input. The underscores are present because the pedaling function does not depend on the second item in the state, which happens to be velocity, and because air resistance does not depend on the first item in the state, which happens to be time.

A list of pairs is something we can plot with the `plotPath` function from the `gnuplot` library, but we need to truncate the list to a finite list before plotting, or `plotPath` will hang while trying to finish calculating an infinite list. In `pedalCoastAirGraph` below, we use the `takeWhile` function to extract the states with times less than or equal to 100 seconds.

```
pedalCoastAirGraph :: IO ()
pedalCoastAirGraph
    = plotPath [Title "Pedaling and coasting with air"
               ,XLabel "Time (s)"
               ,YLabel "Velocity of Bike (m/s)"
               ,PNG "pedalCoastAirGraph.png"
               ,Key Nothing
               ] (takeWhile (\(t,_) -> t <= 100)
                  pedalCoastAir)
```

This code produces Figure 14-13, which shows the velocity as a function of time for the child pedaling and coasting in the presence of air resistance.

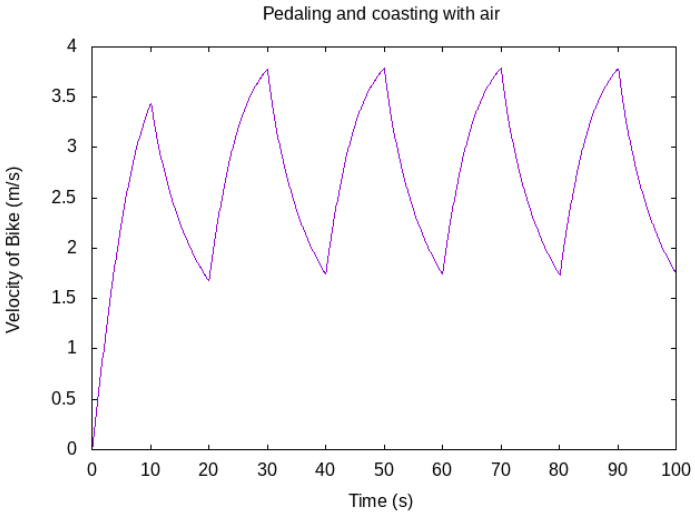

Figure 14-13: Pedaling and coasting with air resistance

As expected, the child's speed increases during the pedaling intervals and decreases during the coasting intervals.

Method 2: Produce a Velocity Function

Now let's use the function `velocityFtv` to produce a velocity function for the child on the bike.

```
pedalCoastAir2 :: Time -> Velocity
pedalCoastAir2 = velocityFtv 0.1 20 (0,0)
                 [\( t,_v) -> pedalCoast t
                 ,\(_t, v) -> fAir 1 1.225 0.5 v]
```

The data we give to `pedalCoastAir2` is the same data we gave to `pedalCoastAir`. Because `pedalCoastAir2` is a function `R -> R`, it can be plotted with the `plotFunc` function from the `gnuplot` package. It would produce the same graph as that in Figure 14-13.

Summary

This chapter discussed Newton's first law and introduced Newton's second law in the context of one-dimensional motion. The chapter presented a sequence of increasingly sophisticated settings for Newton's second law. Easiest among them is when the forces on an object are constant, that is, unchanging in time. Next is when the forces on an object depend only on time, in which case we can apply integration to find the velocity and the position of the object. Forces that depend on velocity, such as the air resistance introduced in this chapter, require that we solve a differential equation, which is a more complex task than integration. The chapter also introduced the Euler method for solving a first-order differential equation. The Euler method, along with Newton's second law, provides a rule for updating the state of the object we are tracking, allowing us to predict its future motion. The choice of state variables, or physical quantities contained in the state, is determined by what the forces depend on. If forces depend only on velocity, then velocity alone can serve as the particle state. If forces depend on time and velocity, we use time and velocity as state variables.

In the next chapter, we allow the forces to depend on position as well as time and velocity. This produces a second-order differential equation and requires that time, position, and velocity all be state variables.

Exercises

Exercise 14.1. Write a function `velocityCF'` that does the same thing and has the same type signature as `velocityCF`, but in which the time `t :: Time` is listed explicitly on the left of the equal sign in the definition.

```
velocityCF' :: Mass
            -> Velocity          -- initial velocity
            -> [Force]           -- list of forces
            -> Time -> Velocity  -- velocity function
velocityCF' m v0 fs t = undefined m v0 fs t
```

Exercise 14.2. Using the `positionCF` function, make a graph for the position of the car on the air track in Example 14.1 as a function of time. Assume the initial position of the car is −1 m.

Exercise 14.3. Write a function

```
sumF :: [R -> R] -> R -> R
sumF = undefined
```

that adds a list of functions to produce a function that represents the sum. Replace `undefined` with your code, and feel free to include one or two variables to the left of the equal sign in the definition. Using `sumF`, we could write the first line in the `let` construction of `velocityFt` as `fNet = sumF fs`.

Exercise 14.4. Write a Haskell function

```
positionFv :: R                    -- time step
           -> Mass
           -> Position             -- initial position x(0)
           -> Velocity             -- initial velocity v(0)
           -> [Velocity -> Force]  -- list of force functions
           -> Time -> Position     -- position function
positionFv = undefined
```

that returns a position function for a Newton's second law situation in which the forces depend only on the velocity. Replace the `undefined` with your code, and feel free to include variables to the left of the equal sign in the definition.

Exercise 14.5. Any Newton's second law problem that can be solved with `velocityFv` can also be solved with `velocityFtv`. Rewrite the `bikeVelocity` function so that it uses `velocityFtv` instead of `velocityFv`.

Exercise 14.6. Doing the Euler method by hand on page 225, we found the velocity after 1.5 s to be $v(1.5\text{ s}) = 2.1295\text{ m/s}$. Use the `velocityFv` function to calculate this same number.

Exercise 14.7. Doing the Euler method by hand on page 235, we found the velocity after 18 s to be $v(18\text{ s}) = 0.8752\text{ m/s}$. Use `statesTV` or `velocityFtv` to calculate this same number.

Exercise 14.8. Fix the stair-stepping issue in Figure 14-9 so that a smooth plot appears.

Exercise 14.9. Write a Haskell function

```
positionFtv :: R                     -- time step
            -> Mass
            -> Position              -- initial position x(0)
            -> Velocity              -- initial velocity v(0)
            -> [(Time,Velocity) -> Force]  -- force functions
            -> Time -> Position      -- position function
positionFtv = undefined
```

that returns a position function for a Newton's second law situation in which the forces depend only on time and velocity. Replace the `undefined` with your code, and feel free to include variables to the left of the equal sign in the definition.

Exercise 14.10. Produce a graph of position versus time for the situation in Figure 14-13.

Exercise 14.11. To deepen our understanding of the Euler method, we'll do a calculation by hand (using only a calculator and not the computer).

Consider a 1-kg mass exposed to two forces. The first force is an oscillatory force, pushing first one way and then the other. With t in seconds, the force in Newtons is given by

$$F_1(t) = 4\cos 2t$$

The second force is an air resistance force in Newtons, given by

$$F_2(v_0) = -3v_0$$

where v_0 is the current velocity of the mass in meters per second.

The net force is

$$F^{tv}_{\text{net}}(t, v_0) = F_1(t) + F_2(v_0) = 4\cos 2t - 3v_0$$

Suppose the mass is initially moving 2 m/s so that

$$v(0\text{ s}) = 2\text{ m/s}$$

Use the Euler method with a time step of $\Delta t = 0.1$ s to approximate the value of $v(0.3\text{ s})$. Keep at least four figures after the decimal point in your calculations. Show your calculations in a small table.

Exercise 14.12. Write a Haskell function

```
updateExample :: (Time,Velocity)  -- starting state
              -> (Time,Velocity)  -- ending state
updateExample = undefined
```

that takes a time-velocity pair (t_0, v_0) and returns an updated time-velocity pair (t_1, v_1) for a single step of the Euler method for a 1-kg object experiencing a net force of

$$F^{tv}_{\text{net}}(t, v_0) = F_1(t) + F_2(v_0) = 4\cos 2t - 3v_0$$

Use a time step of $\Delta t = 0.1$ s. Show how to use the function `updateExample` to calculate the value $v(0.3\text{ s})$ that you calculated by hand in Exercise 14.11.

Exercise 14.13. Consider a 1-kg object experiencing a net force

$$F^{tv}_{\text{net}}(t, v_0) = -\alpha v_0$$

where $\alpha = 1$ N s/m, subject to the initial condition $v(0\text{ s}) = 8$ m/s. Use the Euler method to find the velocity of the object over the time interval $0\text{ s} \leq t \leq 10\text{ s}$. Plot velocity as a function of time to see what it looks like. Compare your results to the exact solution:

$$v(t) = (8\text{ m/s})e^{-\alpha t}$$

Try out different time steps to see what happens when the time step gets too big.

Find a time step that is small enough that the Euler solution and the exact solution nicely overlap on a plot. Find another time step that is big enough that you can see the difference between the Euler solution and the exact solution on a plot.

Make a nice plot (with title, axis labels, and so on) with these three solutions on a single graph (bad Euler, good Euler, and exact). Label the Euler results with the time step you used and label the exact result "Exact."

Exercise 14.14. Consider the differential equation

$$\frac{dv(t)}{dt} = \cos(t + v(t))$$

subject to the initial condition $v(0) = 0$. This differential equation has no exact solution. Use the Euler method with a step size of $\Delta t = 0.01$ to find $v(t)$ over the interval $0 \leq t \leq 3$. Make a nice plot of the resulting function and include the value $v(3)$ to five significant figures.

Exercise 14.15. Each wire in a functional diagram can be labeled with a type. Label each wire in Figure 14-11 with a type.