Physics and Functional Programming

Scott N. Walck

Version 0.1, August 17, 2018

Contents

To Peggy, Carl, Dan, and Jodi.

Preface

One of the best ways to learn something is to teach it. It is invaluable to have a person who is willing to listen to what we say, to read what we write, and to respond. Knowing that someone is listening or reading encourages us to spend time and effort creating something of quality. And if our writing incites a response, so much the better, for we have started a conversation that might challenge us to sharpen our understanding.

A pleasant and productive way to learn physics is to teach a computer how to do it. We admit up front that the computer is not as rich a listener as a person, and cannot provide the depth or breadth of response to our writing that a person can. On the other hand, the computer is very attentive, willing to listen incessantly, and unwilling to accept statements unless they are expressed in clear language and make sense. The computer can provide us with a useful response, because it will happily calculate what we ask it to calculate, and it will quickly tell us if what we just said makes no sense (and hopefully give us a clue about why it makes no sense).

This book is about learning basic theoretical physics by teaching a computer how to do it. We will spend a substantial amount of time with Newton's Second Law. We will focus on the concept of the *state* of a physical system, and see that Newton's Second Law is the core rule for describing how the state changes in time. We will study basic electromagnetic theory, asking the computer to calculate electric and magnetic fields produced by charge and current distributions. The point is to deepen your understanding of physics by approaching it from a new angle, with a new language. The language we will use is precise, and will help to clarify thinking.

This book grows out of a sophomore-level computational physics course that I teach at Lebanon Valley College. Students enrolled in the course have previously taken a year of calculus-based introductory physics. No previous programming experience is assumed.

Since the book begins with a self-contained introduction to the Haskell programming language for people who have not programmed before, it can be used as a supplement for introductory and intermediate courses in physics in which the instructor or student

(a) wishes to include a computational component, or

(b) desires a deeper understanding of the structure of basic physical theories.

The book is also appropriate for self-study by any student who wishes to deepen their understanding of physics by programming.

Physics can be encoded in any programming language. Why use a functional language instead of a more mainstream object-oriented language, say? Beauty and power are to be found more in verbs than in nouns. Newton found beauty and power not in the world per se, but in the description of how the world changes. Functional programming found beauty and power not in objects, but in the functions that take objects as input and produce objects as output, and in the notion that such objects might themselves be functions. Haskell is a good programming language for learning physics for two reasons. First, Haskell is a functional programming language. This means that functions play a central role in the language, including functions that take other functions as arguments and return functions as results. A number of physical ideas are naturally expressed in the language of higher-order functions. Second, Haskell's type system provides a clean way to organize our thinking about the physical quantities and procedures of interest in physics. I know of no better way to clarify my thinking than expressing my ideas in functional language.

This book has been (and continues to be) a labor of love, meaning that my motivation for writing it comes from a love of the ideas and a desire to share them. I am committed to incremental improvement of the text, and I am grateful to hear about and correct any mistakes that the reader finds here. Enjoy!

Part I

Language

In this part, we introduce the Haskell programming language and a few libraries that will form the basic linguistic building blocks in which we will write our ideas.

We will see that Haskell (and even the relatively small subset of Haskell that we will discuss and use) is a rich language with structures that allow and encourage specialization toward a particular domain of interest (in our case, physics). When augmented with a number of libraries, we will have a toolkit capable of expressing most all of the important ideas in theoretical physics, calculating quantities of interest, and presenting results in useful ways (graphs, figures, animations).

Part I draws on physics for examples of the use of language features, but stops short of attempting any systematic expression of physical ideas. The principle focus is on the language itself.

Chapter 1

Introduction to Haskell

In this book, we will use the Haskell programming language to describe calculations that we want the computer to do for us. Haskell is a lovely, deep language, and we will use only a small subset of the language. Studying the language and using it for physics will clarify your thinking.

Haskell is a *functional programming language*, meaning that computations are built out of functions. It encourages a style of programming that is quite different from "imperative" languages like C, C++, Java, Python, and others. As there are no programming prerequisites for this book, we will not spend any more time comparing Haskell to other programming languages. We will just jump in and start learning Haskell.

The Haskell compiler that we will use in this book is the Glasgow Haskell Compiler (GHC). There is an interactive version of the compiler that will be of great use to us called GHCi. We begin by exploring some basic things we can do with GHCi.

1.1 Using GHCi as a calculator

GHCi is the interactive version of the Glasgow Haskell Compiler. It is interactive in the sense that we can enter an expression, and GHCi will evaluate the expression and return a result.

The method of starting GHCi may depend on the operating system your computer uses. Typically you can click on an icon, choose GHCi from a menu, or type `ghci` at a command line.

When GHCi starts, you get a prompt at which you can enter expressions.

The first prompt you get from GHCi is

```
Prelude>
```

The Prelude is a collection of constants, functions, and operators that are available by default, and that you can immediately use to construct expressions. GHCi indicates that the Prelude has been loaded for you by including the name `Prelude` in the prompt.

GHCi is now waiting for you to enter an expression. If you type `2/3`, follow by enter, GHCi will evaluate this expression and print a result.

GHCi
```
2/3
    ⇝  0.6666666666666666
```

In the expression `2/3`, Haskell interprets the `2` and `3` as numbers, and the `/` as a binary operator for division. GHCi performs the requested division and returns the result.

1.2 Numeric functions

Haskell provides functions in the Prelude to do many of the things that you expect calculators to be able to do.

GHCi
```
log(2)
    ⇝  0.6931471805599453
```

This is the natural logarithm function applied to the number 2. The Haskell language does not need parentheses to apply a function. Function application (also known as function use or function evaluation) is such a basic idea in Haskell that the juxtaposition of two expressions is taken to mean that the first expression is a function, the second is an argument, and the function is applied to the argument. Therefore, we can type the following.

GHCi
```
log 2
    ⇝  0.6931471805599453
```

A list of numeric functions available in the Prelude is given in Table 1.1. Haskell provides the constant π in the Prelude.

GHCi
```
pi
    ⇝  3.141592653589793
```

Here is a trigonometric function.

Function	Description
`exp`	$\exp x = e^x$
`sqrt`	square root
`abs`	absolute value
`log`	natural logarithm (log base e)
`sin`	argument in radians
`cos`	argument in radians
`tan`	argument in radians
`asin`	arcsine (inverse sine)
`acos`	arccosine
`atan`	arctangent
`sinh`	$\sinh x = (e^x - e^{-x})/2$
`cosh`	$\cosh x = (e^x + e^{-x})/2$
`tanh`	$\tanh x = (e^x - e^{-x})/(e^x + e^{-x})$
`asinh`	inverse hyperbolic sine
`acosh`	inverse hyperbolic cosine
`atanh`	inverse hyperbolic tangent

Table 1.1: Some common numeric functions

GHCi
```
cos pi
    ⇝  -1.0
```

Notice that trigonometric functions in Haskell expect an argument in radians.

Let's calculate $\cos\frac{\pi}{2}$.

GHCi
```
cos pi/2
    ⇝  -0.5
```

The computer did not give us what we expect here; $\cos\frac{\pi}{2} = 0$, not $-\frac{1}{2}$. The reason is that function application in Haskell has higher precedence than division, so Haskell interprets what we typed as

GHCi
```
(cos pi)/2
    ⇝  -0.5
```

rather than dividing π by two first and then taking the cosine. We can get what we want by supplying parentheses.

GHCi
```
cos (pi/2)
    ⇝  6.123233995736766e-17
```

Is the result the computer gave $\cos\frac{\pi}{2}$? Not exactly. Here we see an example of an approximately computed result. My computer gave something times 10^{-17}, which is as close to zero as the computer can get here. It's good to remember that when doing numerical work, the computer (like your calculator) is not giving us exact results most of the time. It is giving us approximate results. We need to be vigilant in making sure that the results it gives are valuable to us by interpreting them correctly. Later, after we discuss Haskell's system of types, we will say more about when we can and when we cannot expect the computer to produce exact results.

1.3 Operators

The Haskell Prelude provides a number of infix operators, shown in Table 1.2. They are called "infix" because they sit between the two items that they act on. (In computer science generally, an operator placed before its arguments is called a prefix operator, and an operator placed after its arguments is called a postfix operator. In Haskell, operators always mean infix operators.)

Table 1.2 shows operators for addition, subtraction, multiplication, and division. These work pretty much as you would expect. The table shows three different operators for exponentiation. This proliferation is related

Operation	Operator	Precedence	Associativity
Composition	`.`	9	Right
Exponentiation	`^`, `^^`, `**`	8	Right
Multiplication, division	`*`, `/`	7	Left
Addition, subtraction	`+`, `-`	6	Left
List operators	`:`, `++`	5	Right
Equality, Inequality	`==`, `/=`	4	
Comparison	`<`, `>`, `<=`, `>=`	4	
Logical And	`&&`	3	Right
Logical Or	`||`	2	Right
Application	`$`	0	Right

Table 1.2: Precedence and associativity for common operators

to Haskell's type system, about which we will say more later. The "carrot" operator `^` can handle only nonnegative integer exponents. The double carrot `^^` can handle any integer exponent. The `**` can handle any real exponent. For now, I recommend `**` for exponentiation.

The equality, inequality, and comparison operators can be used between numeric expressions.

GHCi
```
pi > 3
    ⇝  True
```

The result of a comparison is a Boolean expression (`True` or `False`).

We will discuss the other operators in Table 1.2 at a later point.

As we saw earlier when we tried to take the cosine of $\pi/2$, function application takes precedence over infix operators. In addition, some operators take precedence over other operators.

In the expression

GHCi
```
1 + 2 * 3
    ⇝  7
```

the multiplication of 2 and 3 will occur before the addition with 1. This is consistent with usual mathematical notation. In order to carry this out, binary operators in Haskell have a precedence associated with them that describes which operations should be carried out first. Binary operators have a precedence from 0 to 9. A higher precedence means an operation will be carried out first. For example, addition and subtraction have a precedence of 6 in Haskell, while the precedence of multiplication and division is 7 and the precedence of exponentiation is 8. The "or" operation between Boolean values has a precedence of 2, and the "and" operation has a precedence of 3.

The far right column of Table 1.2 lists associativity of some operators. Consider the expression `8 - 3 - 2`. There are two ways in which this expression might be interpreted. One interpretation (and this is the standard mathematical convention) is that the expression is shorthand for $(8-3)-2$, which evaluates to 3. Another interpretation is that the expression is shorthand for $8-(3-2)$, which evaluates to 7. Clearly, it is important for us to understand which of these two interpretations is correct for the original expression, and that is where associativity comes in. Looking at Table 1.2, we see that subtraction is left associative. This means that the left-most subtraction is carried out first, resulting in the first interpretation given above (resulting in 3, not 7).

Precedence and associativity allow us to unambiguously determine which operators act first.

Example 1.1. Add parentheses to the following expression to indicate the order in which Haskell's precedence and associativity rules would evaluate the expression.

```
8 / 7 / 4 ** 2 ** 3 > sin pi/4
```

Solution: Function application takes precedence over all operators, so `sin pi` is the first thing calculated.

```
8 / 7 / 4 ** 2 ** 3 > (sin pi)/4
```

Next, exponentiation is the operator in the expression with the highest precedence in Table 1.2. Exponentiation occurs twice. Since it is right associative, the right-most exponentiation occurs next.

```
8 / 7 / 4 ** (2 ** 3) > (sin pi)/4
```

Next is the left exponentiation.

```
8 / 7 / (4 ** (2 ** 3)) > (sin pi)/4
```

Next is division. There are three divisions. The right-most division is unproblematic, but the two divisions on the left of the expression need to be resolved by associativity rules. Division is left associative.

```
(8 / 7) / (4 ** (2 ** 3)) > ((sin pi)/4)
```

Note we have inserted two sets of parentheses in the last step. One is for the right-most division and one is for the left-most division. Now we can put parens in for the inner division.

```
((8 / 7) / (4 ** (2 ** 3))) > ((sin pi)/4)
```

The last operator to act is the comparison operator `>`. There is no need to put parens around the entire expression, so we are done. The fully parenthesized expression is

```
((8 / 7) / (4 ** (2 ** 3))) > ((sin pi)/4)
```

The purpose in learning the precedence and associativity rules is so that we can avoid using parentheses as much as possible. Multiple levels of nested expressions make things hard to read. Try never to use more than two levels of nested parentheses. In addition to knowing the precedence and associativity rules, there are other ways to avoid the use of parentheses, such as defining a local variable, that we will discuss later.

1.4 Numbers in Haskell

1.4.1 Negative numbers in Haskell

If you try

```
5 * -1
```

you will get an error, although the meaning of the expression seems clear enough; we want to multiply 5 by -1. The trouble here is that the minus sign acts as both a binary operator (as in the expression $3 - 2$) and a unary operator (as in the expression -2). Binary operators play an important role in Haskell, and the syntax of the language supports their use in a consistent, unified way. Unary operators in Haskell are much more of a special case; in fact, the minus sign is the only one. Because of decisions the Haskell designers made (which seem in hindsight to have been good decisions), negative numbers are sometimes not recognized as readily as you might expect.

The solution is simply to enclose negative numbers in parentheses. For example,

GHCi
```
5 * (-1)
    ⇝  -5
```
evaluates readily to −5.

1.4.2 Decimal numbers in Haskell

A number containing a decimal point must have digits (0 through 9) both before and after the decimal point. Thus we must write `0.1` not `.1`; instead of `5.` we must write either `5.0` or `5` without a decimal point. The reason is that the dot character serves another role in the language (namely function composition, mentioned in the top line of Table 1.2, which we will study later). The rule requiring digits before and after a decimal point helps the compiler distinguish the meaning of the dot character.

1.4.3 Exponential notation

You can use exponential notation to describe numbers in Haskell that are very big or very small. Here are a few examples.

Mathematical notation	Haskell notation
3.00×10^{8}	`3.00e8`
6.63×10^{-34}	`6.63e-34`

Haskell will also use exponential notation to show you numbers that turn out to be very big or very small.

Expression		evaluates to
`8**8`	⇝	`1.6777216e7`
`8**(-8)`	⇝	`5.960464477539063e-8`

1.5 Functions with two arguments

All of the functions in Table 1.1 take one real number as input and give a real number as output (assuming the input is in the domain of the function). There are a couple of useful numeric functions that take two real numbers as input. These are listed in Table 1.3.

Function	Example
`logBase`	`logBase 10 100` $= 2$
`atan2`	`atan2 1 0` $= \pi/2$

Table 1.3: Numeric functions with two arguments

GHCi
```
logBase 10 100
    ⇝  2.0
```
GHCi
```
atan2 1 0
    ⇝  1.5707963267948966
```

The `logBase` function takes two arguments; the first is the base of the logarithm and the second is the number we wish to take the log of.

The `atan2` function solves a problem you may have run into if you've tried to use the inverse tangent function to convert from Cartesian to polar coordinates. Consider the following equations for polar coordinates (r, θ) in terms of Cartesian coordinates (x, y).

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \tan^{-1}(y/x)$$

Suppose we are trying to find the polar coordinates associated with the point $(x, y) = (-1, -\sqrt{3})$. The answer needs to be a point in the 3rd quadrant, since x and y are both negative. This means θ should be in the range $\pi < \theta < 3\pi/2$ (or $-\pi < \theta < -\pi/2$). But if we mechanically apply the formula above for θ, we will calculate $\tan^{-1}(\sqrt{3})$, which our calculator or computer will tell us is $\pi/3$. The problem is the domain of the inverse tangent function, and a solution is to use the `atan2` function instead of the `atan` function. The result of `atan2` y x will give the angle, in radians, in the correct quadrant.

Note how the two arguments are given to the functions `logBase` and `atan2`. In particular, there is no comma between the two argument values, as would be required in traditional mathematical notation.

1.6 Approximate calculation

Most of the calculations that we will do are not exact calculations. When we ask the computer to find the square root of 5,

GHCi
```
sqrt 5
    ⇝  2.23606797749979
```

it gives a very accurate result, but it is not an exact result. The computer uses a finite number of bits to represent this number.

If you evaluate `sqrt 5 ^ 2` in GHCi, you may not get exactly 5.0 as a result.

GHCi
```
sqrt 5 ^ 2
    ⇝  5.000000000000001
```

The computer does not represent $\sqrt{5}$ exactly. We can even ask the following in GHCi.

GHCi
```
sqrt 5 ^ 2 == 5
    ⇝  False
```

My computer gives `False`, because of the approximate calculation.

Another source of non-exactness in calculation comes from the computer's use of a binary (base 2) internal representation of numbers. When I multiply $3 * 0.2$, I don't get exactly 0.6. Why? The reason is that 0.2, which has a nice finite decimal (base 10) representation, is a repeating binary (base 2) number. Just like the fraction 1/3 has an infinite repeating representation in base 10 (0.333333...), the fraction 1/5 has an infinite repeating representation in base 2.

Number	Decimal	Binary
1/2	0.5	0.1
1/3	0.333333...	0.01010101...
1/4	0.25	0.01
1/5	0.2	0.001100110011...

The computer converts every number that we supply in decimal into its internal binary form, and only keeps a finite number of digits (bits, really). Most of the time, we don't need to be concerned with this, but it explains why some calculations that seem like they should be exactly calculable are not. The moral of this story is never to do equality checking of numbers when either number has been approximately calculated.

1.7 Errors

People make mistakes. This is as it should be. When you enter something the computer does not understand, it will give you an error message. These messages can appear intimidating, but they are a great opportunity for learning, and it is worthwhile to learn how to read them.

1.7.1 Variable not in scope

One of the simplest types of error comes from using a variable that has not been defined. If we ask GHCi for the value of `x` without having defined `x`, we will get a ”variable not in scope” error.

GHCi
```
x
```
$\rightsquigarrow$

```
<interactive>:19:1: error: Variable not in scope: x
```

The *scope* of a variable is the set of situations in which it can be used. The idea of scope is important and more complex than just defining or not defining something. For now, ”variable not in scope” means that you used a variable, the computer expected to already know what that variable stood for, but didn’t.

1.7.2 No instance for Show

There are some completely legitimate, well-defined objects in Haskell that have no good way of being shown on the screen. Functions are the most common example. Since a function can accept a wide variety of inputs and produce a wide variety of outputs, there is in general no good way of displaying the ”value” of a function. If you ask GHCi to tell you what the square root function ”is”, it will complain that it knows no way to show it to you by saying that is has ”no instance of Show”.

GHCi
```
sqrt
```
$\rightsquigarrow$

```
<interactive>:20:1: error:
    No instance for (Show (Double -> Double))
      arising from a use of print
```

```
        (maybe you haven't applied a function to enough arguments?)
    In a stmt of an interactive GHCi command: print it
```

This message is not due to an error at all. GHCi is merely telling you that it can't display the thing you want.

1.8 Getting help and quitting

To ask GHCi for help type `:help` (or `:h`). To leave GHCi, type `:quit` (or `:q`).

Commands that start with a colon do not belong to the Haskell programming language proper, but rather to the GHCi interactive compiler. We will see more of these commands that start with colons later.

1.9 More information

To learn more about the Haskell programming language (of which GHCi is a popular implementation), you can visit the web site `http://www.haskell.org/`.

The `haskell.org` web site has links to a number of online and paper sources for learning the language. Some particularly good ones are *Learn You a Haskell for Great Good* (`http://learnyouahaskell.com/`) and *Real World Haskell* (`http://book.realworldhaskell.org/`).

1.10 Exercises

Exercise 1.1. Evaluate `sin 30` in GHCi. Why is it not equal to 0.5?

Exercise 1.2. Add parentheses to the following expressions to indicate the order in which Haskell's precedence and associativity rules would evaluate the expressions.

(a) `2 ^ 3 ^ 4`

(b) `2 / 3 / 4`

(c) `7 - 5 / 4`

(d) `log 49/7`

Exercise 1.3. Use GHCi to find $\log_2 32$.

Exercise 1.4. Use the `atan2` function in GHCi to find the polar coordinates (r, θ) associated with Cartesian coordinates $(x, y) = (-3, 4)$.

Exercise 1.5. Find a new example of a calculation in which the computer produces a result that is just a little bit different from the exact result.

Exercise 1.6. Why is there no associativity listed for the equality, inequality, and comparison operators in Table 1.2? (Hint: Write down the simplest expression you can think of that would require the associativity rules to resolve the precedence of comparison operators and try to make sense of it.)

Chapter 2

Functions

Programming in Haskell is a process of defining functions that express to the computer how to calculate something we want. There is a way to define functions inside GHCi, but since most functions that we define we will want to use again another day, it is better to define our functions in a file, and load that file into GHCi.

We need a text editor to create such a file. Examples of popular text editors are GNU Emacs and `gedit`. Word-processing programs that you might use to type a letter or a document are not appropriate for this purpose, because they store the text you type with additional information (such as font type and size) that will make no sense to the Haskell compiler.

2.1 Constants, functions, and types

Using a text editor, let's create a file named `first.hs` for our first program. (The `.hs` extension indicates a Haskell program.) Let's put the following lines in the file.

```haskell
-- First Haskell program

-- Here we define a constant
e :: Double
e = exp 1

-- Here we define a function
```

```
square :: Double -> Double
square x = x**2
```

This program file defines a constant and a function. The lines that begin with `--` are *comments*. The Haskell compiler ignores any line that begins with a double hyphen; the purpose of a comment is to aid a human reader of the code. Code samples in this book highlight comments in orange. Your text editor may or may not highlight the text you type. The Haskell compiler does not see this colorization. Color is used in this book in an effort to make the purpose of the code easier to understand.

The first two non-comment lines of the file define the constant e, the base of natural logarithms. Unlike π, e is not included in the Haskell Prelude. The line

```
e :: Double
```

declares the *type* of `e` to be `Double`. A `Double` is an approximation to a real number, sometimes called a floating point number. The name `Double` is used for historical reasons to mean a *double precision* floating point number. This type of number is capable of about 15 decimal digits of precision, compared with a single precision number that is capable of about 7 decimal digits of precision. The difference for the computer is the number of bits used to represent the number inside the computer. Haskell has a type `Float` for single precision numbers. Unless there is a compelling reason to do otherwise, we will always use type `Double` for our (approximations to) real numbers. Types, such as `Double` and `Float`, are displayed in blue in this book.

In addition to `Double`, there are a number of other types that we might want to use. Haskell has a type `Int` for small integers (up to at least a few billion), and a type `Integer` for arbitrary size integers.

Let's get back to our `first.hs` program file. As we said above, the first line of the file declares the type of the name `e` to be `Double`. This kind of line, with a name followed by a double colon followed by a type, is called a *type signature*. We may also call such a line a *declaration*, because it declares the name `e` to have type `Double`.

The second line of the file actually *defines* `e`. Here, we use the built-in function `exp` applied to the number `1` to produce the constant `e`. Remember that we don't need parentheses to apply a function to an argument.

Next, we have a type signature for the function `square`. The type of `square` is declared to be `Double -> Double`. A type containing an arrow is called a *function type*. (Function types will be explored in more detail in section 3.2.) It says that `square` is a function that takes a `Double` as input and produces a `Double` as output. The last line defines the function `square`. Note the `**` operator used for exponentiation.

To load this program file into GHCi, use GHCi's `:load` command. (`:l` for short).

```
GHCi  :l first.hs
```

Note that the GHCi prompt changes from `Prelude>` to `*Main>`. This indicates that our program file has been successfully loaded and given the default name `Main`. You now have access to the constant and function defined in the file. Let's try them out.

```
GHCi  square 7
          ⇝  49.0
GHCi  square e
          ⇝  7.3890560989306495
```

The names `e` and `square` defined in the file `first.hs` are examples of *variable identifiers* in Haskell. Variable identifiers must begin with a lowercase letter, followed by zero or more uppercase letters, lowercase letters, digits, underscores, and single quotes. Names that begin with an uppercase letter are reserved for types (which we discuss in chapter 3), type classes (which we discuss in chapter 7), and module names.

If you forget or don't know the type of something, you can ask GHCi for the type with the `:type` command (`:t` for short).

```
GHCi  :t square
          ⇝  square :: Double -> Double
```

The notation used for defining a function in Haskell is similar to mathematical notation in some ways, and different in a few ways. Let's comment on the differences. Examples are shown in Table 2.1.

1. Traditional mathematical notation (and some computer algebra systems) use juxtaposition to represent multiplication. For example, $2x$ means 2 multiplied by x, just because the symbols are next to each other. Haskell requires use of the multiplication operator `*`. In Haskell,

juxtaposition means function application.

2. Traditional mathematical notation requires that function arguments be put in parentheses after the function name. This is true for function definitions (compare $f(x) = x^3$ with Haskell's `f x = x**3`) as well as function applications (compare $f(2)$ with Haskell's `f 2`). Haskell does not require parentheses in function definition or application. Haskell uses parentheses to indicate the order of operations.

3. Traditional mathematical notation tries to get away with single-letter function names, such as f. Haskell allows single-letter function names, but it is much more common to use a multi-letter word for a function name (such as `square` above), especially when the word can serve as a good description of what the function does.

Mathematical definition	Haskell definition
$f(x) = x^3$	`f x = x**3`
$f(x) = 3x^2 - 4x + 5$	`f x = 3 * x**2 - 4 * x + 5`
$g(x) = \cos 2x$	`g x = cos (2 * x)`
$v(t) = 10t + 20$	`v t = 10 * t + 20`
$h(x) = e^{-x}$	`h x = exp (-x)`

Table 2.1: Comparison of function definitions in traditional mathematical notation with function definitions in Haskell

2.2 How we talk about functions

Suppose we define a function f by $f(x) = x^2 - 3x + 2$. It is common in mathematics and physics to speak of "the function $f(x)$." Haskell invites us to think a bit more carefully and precisely about this bad habit. (Well, it really *requires* us to think more carefully about it, but it's always nicer to be *invited* than required, no?) Instead of saying "the function $f(x)$," we should say one of the following, depending on what we mean.

- the function f
- the value $f(x)$

- given a number x, the value of the function f at x

The second and third bullet points are two ways of saying the same thing. The first bullet point is saying something different from the second and third.

What is wrong with saying "the function $f(x)$?" It is common in mathematics and physics to use "the function f" and "the function $f(x)$" interchangeably, the second expression merely indicating explicitly that f depends on x. We think of mathematical notation as being a precise representation of an idea, but this is a case where the commonly used notation is not precise.

When we use Haskell we make a trade off. We agree to use language in a precise and careful way (the compiler is going to check us on this) and in exchange, we will be able to say things in the language that (1) are rather complex, (2) are difficult to say in a language that accommodates imprecision, and (3) expose the essential structure of a physical theory like Newtonian mechanics.

One reason for shunning the language "the function $f(x)$" is that if $f(x) = x^2 - 3x + 2$, then $f(y) = y^2 - 3y + 2$. The letter x really has nothing to do with the function f. Granted, we need *some* letter to use to make the definition, but it doesn't matter which one.

In Haskell, we say $f(x)$ when we wish to evaluate the function f using the input x. We say f when we wish to speak about the function itself, not evaluating it (not giving it any input). What else is there to do with a function except give it an input? Well, you could *integrate* the function between given limits. You could *differentiate* the function to obtain another function. You could, in some cases, apply the function twice. In short, there are many things we might want to do with a function other than simply evaluating it.

Haskell's type system helps us understand the key distinction between f and $f(x)$. The variable x is a number, so has a type like `Double`. Now f is a function, so has a type like `Double -> Double`. Finally, $f(x)$ means the function f evaluated at the number x, so $f(x)$ has type `Double`. Things that have type `Double -> Double` are functions. Things that have type `Double` are numbers. The table below summarizes these distinctions.

Math notation	Haskell notation	Haskell type
f	`f`	`Double -> Double`
$f(3)$	`f 3`	`Double`
$f(x)$	`f x`	`Double`

Computers are notorious for being inflexible in understanding what a person means. Computers look at exactly what you said, and give warnings and errors if your input doesn't meet its requirements for format and interpretation. Much of the time, this is a pain in the neck. We would like to have an assistant that understanding what we mean, and does what we want.

In the case of types and functions, however, Haskell's inflexibility is a great teaching aid. Haskell is helping us to organize our thinking, so that we will be prepared to do more complex things in a structured and organized way. Section 5.1 on higher-order functions is an example of how careful thinking about types and functions allows us to encode more complex ideas simply and easily.

2.3 Anonymous functions

Haskell provides a way to specify a function without naming it. For example, the function that squares its argument can be written as follows.

```
\x -> x**2
```

A function specified in this way is called an *anonymous function* or a *lambda function* after the lambda calculus developed by Alonzo Church in the 1930s. (Church was Alan Turing's Ph.D. advisor.) The backslash character (\) was thought by Haskell's creators to look a bit like the lowercase greek letter lambda (λ).

Table 2.2 shows examples of mathematical functions written as lambda functions. This is an alternative way to define the functions in Table 2.1. The real power of lambda functions, however, comes from *not* naming them, instead using them in places where a function is needed, but we don't wish to spend the effort (a declaration and a definition) to name a new function. We will see examples of how this is useful in Chapter 5, where we discuss higher-order functions that take other functions as input. These other functions are sometimes conveniently expressed as anonymous functions.

Mathematical function	Haskell lambda function
$f(x) = x^3$	`f = \x -> x**3`
$f(x) = 3x^2 - 4x + 5$	`f = \x -> 3 * x**2 - 4 * x + 5`
$g(x) = \cos 2x$	`g = \x -> cos (2 * x)`
$v(t) = 10t + 20$	`v = \t -> 10 * t + 20`
$h(x) = e^{-x}$	`h = \x -> exp (-x)`

Table 2.2: Comparison of function definitions in traditional mathematical notation with lambda functions defined in Haskell

We can apply the anonymous squaring function `\x -> x**2` to the argument 3 by writing `(\x -> x**2) 3` at the GHCi prompt.

GHCi
```
(\x -> x**2) 3
    ⇝  9.0
```

Notice that when we write `\x -> x**2`, we are *not* defining what `x` is. Instead we are saying that if we temporarily allow `x` to stand for the argument of the function (such as 3 above), then we have a rule for determining the value of the function applied to the argument. The same remark is true of (named) mathematical functions; when we define $f(x) = x^2$, this is a definition for f, not a definition for x. The function `\x -> x**2` is the same as the function `\y -> y**2`; the variable that we use to name the argument is not important. Both functions are the function that squares its argument. Table 2.3 shows examples of the application of anonymous functions to an argument. These examples could be evaluated at the GHCi prompt.

Expression		evaluates to
`(\x -> x**2) 3`	⇝	`9.0`
`(\y -> y**2) 3`	⇝	`9.0`
`(\x -> x**3) 3`	⇝	`27.0`
`(\x -> 3 * x**2 - 4 * x + 5) 3`	⇝	`20.0`
`(\x -> cos (2 * x)) pi`	⇝	`1.0`
`(\t -> 10 * t + 20) 3`	⇝	`50`
`(\x -> exp (-x)) (log 2)`	⇝	`0.5`

Table 2.3: Examples of applying anonymous functions to an argument

2.4 Exercises

Exercise 2.1. In a Haskell program file (a new file with a new name that ends in `.hs`), define the function $f(x) = \sqrt{1+x}$. As we did for the function `square` above, give both a type signature and a function definition. Then load this file into GHCi and check that $f(0)$ gives 1, $f(1)$ gives about 1.414, and $f(3)$ gives 2.

Exercise 2.2. Consider throwing a rock straight upward from the ground at 30 m/s. Ignoring air resistance, find an expression $y(t)$ for the height of the rock as a function of time.

Add on to your program file `first.hs` by writing a function

```
yRock30 :: Double -> Double
```

that accepts as input the time (after the rock was thrown) in seconds and gives as output the height of the rock in meters.

Exercise 2.3. Continuing with the rock example, write a function

```
vRock30 :: Double -> Double
```

that accepts as input the time (after the rock was thrown) in seconds and gives as output the upward velocity of the rock in meters per second. (A downward velocity should be returned as a negative number.)

Exercise 2.4. Define a function `sinDeg` that computes the sine of an angle given in degrees. Test your function by evaluating `sinDeg 30`.

Exercise 2.5. Write Haskell function definitions for the following mathematical functions. In each case, write a type signature (the type should be `Double -> Double` for each function) and a function definition. You will need to pick alternative names for some of these functions, because Haskell functions must begin with a lowercase letter. Do not use more than two levels of nested parentheses.

(a) $f(x) = \sqrt[3]{x}$

(b) $g(y) = e^y + 8^y$

(c) $h(x) = \dfrac{1}{\sqrt{(x-5)^2 + 16}}$

(d) $\gamma(\beta) = \dfrac{1}{\sqrt{1-\beta^2}}$

(e) $U(x) = \dfrac{1}{10+x} + \dfrac{1}{10-x}$

(f) $L(l) = \sqrt{l(l+1)}$

(g) $E(x) = \dfrac{1}{|x|^3}$

(h) $E(z) = \dfrac{1}{(z^2+4)^{3/2}}$

Exercise 2.6. (a) Express $\gamma(\beta) = \dfrac{1}{\sqrt{1-\beta^2}}$ as an anonymous function.

(b) Write an expression that applies the anonymous function from part (a) to the argument 0.8. What result do you get from GHCi?

Chapter 3

Types

3.1 Basic types

The idea that every expression has a *type* is central to the Haskell programming language. Haskell's most important basic types are shown in Table 3.1. The `Bool` type is for values that are either true or false, like the result of a comparison. For example, `3 > 4` evaluates to `False`.

GHCi
```
3 > 4
    ⇝  False
```

The `Char` type is for single characters. The `String` type is for strings of characters. We have already mentioned the four basic numeric types `Int`, `Integer`, `Float`, and `Double`.

It should be noted that the numeric examples in the right-most column

Type	Description	Examples
`Bool`	Boolean	`False`, `True`
`Char`	character	`'h'`, `'7'`
`String`	string	`"101 N. College Ave."`
`Int`	small integer	`42`
`Integer`	arbitrary integer	`18446744073709551616`
`Float`	single-precision floating point	`0.33333334`
`Double`	double-precision floating point	`0.3333333333333333`

Table 3.1: Haskell's basic types

can be expressions of the type indicated, but that an expression by itself, such as `42`, does not *necessarily* have type `Int`. To be specific, `False` and `True` must have type `Bool`, `'h'` and `'7'` must have type `Char`, and `"101 N. College Ave."` must have type `String`. On the other hand, `42` could have type `Int`, `Integer`, `Float`, or `Double`. Clarifying this ambiguity is one reason to give a type signature with each name you define in a Haskell program. Without a type signature, the compiler cannot tell which of the four numeric types we might want for a number like `18446744073709551616`. Any of the four numeric types would try to hold the number, but only `Integer` would represent the number exactly. The complexity of numeric types in Haskell is related to a more advanced language feature called *type classes* that we will discuss in chapter 7.

3.1.1 The Boolean type

The type `Bool` has only two inhabitants: `False` and `True`. As such, the type is used for expressions that are meant to represent claims that might be true or false.

Haskell has an if-then-else expression whose value depends on a Boolean. The expression has the form `if` b `then` c `else` a. Here b is an expression of type `Bool` called the *condition*, c is called the *consequent*, and a is called the *alternative*. Haskell's type system demands not only that b have type `Bool`, but also that the consequent c and the alternative a have the same type (this can be any type, `Bool` or something else). If the condition b evaluates to `True`, then the entire if-then-else expression evaluates to c; if the condition b evaluates to `False`, then the entire if-then-else expression evaluates to a. In this book, we use bright purple for Haskell keywords such as `if`, `then`, and `else`.

As an example of the if-then-else expression, consider the following function (sometimes called the *Heaviside step function* or the *unit step function*).

$$H(x) = \begin{cases} 0 & , \quad x \leq 0 \\ 1 & , \quad x > 0 \end{cases} \tag{3.1}$$

We can write a definition for this function in Haskell using the if-then-else construction. In Haskell, we are not allowed to begin the names of constants or functions with capital letters (recall the discussion of variable identifiers in section 2.1), so we will call this function `stepFunction`.

```
stepFunction :: Double -> Double
stepFunction x = if x <= 0
                 then 0
                 else 1
```

The function `stepFunction` accepts a `Double` as input (called `x` in the definition), and returns a `Double` as output. The expression `x <= 0` is the condition, the expression `0` is the consequent, and the expression `1` is the alternative.

The Prelude provides a few functions that work with Booleans. The first is `not`, which has type `Bool -> Bool`, meaning it accepts a Boolean as input and gives another Boolean as output. The function `not` returns `True` if its input is `False`, and returns `False` if its input is `True`. You can see this for yourself in GHCi if you type

```
GHCi  not False
         ⇝  True
```

or

```
GHCi  not True
         ⇝  False
```

at the GHCi prompt. GHCi has a command `:type` (`:t` for short) that asks about the type of something. You can ask GHCi for the type of `not` by entering

```
GHCi  :t not
         ⇝  not :: Bool -> Bool
```

at the GHCi prompt. GHCi commands that start with a colon are not part of the Haskell language itself. You cannot use them in a Haskell program file.

The Boolean "and" operator `&&` takes two Booleans as input and gives one Boolean as output. The output is `True` only when both inputs are `True` and `False` otherwise. The behavior of the `&&` operator is described in Table 3.2.

x	y	x `&&` y
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

Table 3.2: Definition of the "and" operator

The Boolean "or" operator `||` takes two Booleans as input and gives one Boolean as output. The output is `False` only when both inputs are `False` and `True` otherwise. The behavior of the `||` operator is described in Table 3.3.

x	y	x `||` y
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Table 3.3: Definition of the "or" operator

These operators are listed in Table 1.2 with their precedence and associativity. You can play with them in GHCi, asking for evaluations of expressions such as

GHCi
```
True || False && True
    ⇝  True
```

at the GHCi prompt.

3.1.2 The character type

The `Char` type is for single characters, including uppercase and lowercase letters, digits, and some special characters (like the newline character that produces a new line of text). Here are some examples of definitions of characters.

```
ticTacToeMarker :: Char
ticTacToeMarker = 'X'

newLine :: Char
newLine = '\n'
```

There is very little reason to make these definitions, since any place where we could use `newLine`, for example, we could just as easily use `'\n'`, which takes up less space. We do it here only to show the relationship between the term `'X'` and the type `Char`. As shown in the examples above, a character can be formed by enclosing a single letter or digit in single quotes.

3.1.3 The string type

A string is a sequence of characters. (In chapter 4, we will learn that a string is a *list* of characters, where list has a precise meaning.) Here are some examples.

```
hello :: String
hello = "Hello, world!"

errorMessage :: String
errorMessage = "Can't take the square root of a Boolean!"
```

These definitions are not as useless as the ones above for characters, because although `"Hello, world!"` is entirely equivalent to `hello`, the name `hello` is at least shorter and easier to type than the string it represents. If such a string was needed at several different places in a program, that would justify the definition of a name such as `hello`. Note that to form a string from a sequence of characters, we enclose the character sequence in double quotes.

3.2 Function types

Haskell provides a number of ways to form new types from existing types. Given any two types `a` and `b`, there is a type `a -> b` for functions that take as input an expression of type `a` and produce as output an expression of type `b`. Here is an example.

```
isX :: Char -> Bool
isX c = c == 'X'
```

The function `isX` takes a character as input and gives a Boolean as output. The function returns `True` if the input character is `'X'`, and `False` otherwise. Adding parentheses may help in reading the function definition. The definition is equivalent to

```
isX c = (c == 'X')
```

In general in a definition, the name on the left of the single equals sign `=` is begin defined (`isX` in this case), and the expression on the right of the single equals sign is the body of the definition. The expression `c == 'X'` uses the equality operator `==` from Table 1.2 to ask if the input character `c` is the same as `'X'`.

If we put this function definition into a Haskell program file (say `FunctionType.hs`) and load it into GHCi,

```
GHCi   :l FunctionType.hs
```

we can ask about the types of things. If we ask about the type of `isX`

```
GHCi   :t isX
           ⇝  isX :: Char -> Bool
```

we see what we wrote in our type signature. In GHCi we can also ask for the type of `isX 't'`.

```
GHCi   :t isX 't'
           ⇝  isX 't' :: Bool
```

This makes sense, because the expression `isX 't'` represents the function `isX` applied to the character argument `'t'`. The type therefore represents the type of the output of `isX`, namely `Bool`.

We can also ask GHCi for the *value* of `isX 't'` (as opposed to the type of the expression). If we enter `isX 't'` at the GHCi prompt,

```
GHCi   isX 't'
           ⇝  False
```

we see that the value of `isX 't'` is `False`, because `'t'` is not equal to `'X'`.

Here is an example of a function with type `Bool -> String`.

```
bagFeeMessage :: Bool -> String
bagFeeMessage checkingBags = if checkingBags
                             then "There is a $100 fee."
                             else "There is no fee."
```

The function `bagFeeMessage` takes a Boolean as input and gives a string as output. The input Boolean (called `checkingBags`) is intended to represent an answer (`True` or `False`) to the question of whether a passenger is checking bags. The style of naming a variable by sticking words together without spaces, using a capital letter for the second and subseqent words, is common in Haskell programming.

There is an alternative way to write the function `bagFeeMessage` that uses a facility in Haskell called *pattern matching*. The type `Bool` allows pattern matching, and other types that we will encounter later also allow pattern matching. The idea behind pattern matching for `Bool` is that the only possible inputs are `False` and `True`, so why not just give the output for each possible input. Here is what the function looks like using pattern matching.

```
bagFeeMessage2 :: Bool -> String
bagFeeMessage2 False = "There is no fee."
bagFeeMessage2 True  = "There is a $100 fee."
```

Notice that by using pattern matching we have avoided using the if-then-else construction. Notice also that we no longer need the variable `checkingBags`, which held the input value.

3.3 Exercises

Exercise 3.1. Add parentheses to the following expressions to indicate the order in which Haskell's precedence and associativity rules (Table 1.2) would evaluate the expressions. Some of the expressions are well-formed and have a clear type. In those cases, give the type of the (entire) expression. Also identify expressions that are not correctly formed (and consequently do not have a clear type) and say what is wrong with them.

(a) `False || True && False || True`

(b) `2 / 3 / 4 == 4 / 3 / 2`

(c) `7 - 5 / 4 > 6 || 2 ^ 5 - 1 == 31`

(d) `2 < 3 < 4`

(e) `2 < 3 && 3 < 4`

(f) `2 && 3 < 4`

Exercise 3.2. Write Haskell function definitions for the following mathematical functions. In each case, write a type signature (the type should be `Double -> Double` for each function) and a function definition.

(a) $f(x) = \begin{cases} 0 & , \quad x \leq 0 \\ x & , \quad x > 0 \end{cases}$

(b) $E(r) = \begin{cases} r & , \quad r \leq 1 \\ \frac{1}{r^2} & , \quad r > 1 \end{cases}$

Exercise 3.3. Define a function `isXorY` with type signature

```
isXorY :: Char -> Bool
```

that will return `True` if the input character is `'X'` or `'Y'` (capital X or Y) and `False` otherwise. Test your function by loading it into GHCi and giving it inputs of `'X'`, `'Y'`, `'Z'`, and so on.

Exercise 3.4. Define a function `bagFee` with type signature

```
bagFee :: Bool -> Int
```

that will return the integer 100 if the person is checking bags and the integer 0 if not. Use an if-then-else construction for this function. Then define a second function `bagFee2` with the same type signature that uses pattern matching instead of the if-then-else construction.

Exercise 3.5. Define a function `greaterThan50` with type signature

```
greaterThan50 :: Integer -> Bool
```

that will return `True` if the given integer is greater than 50, and `False` otherwise.

Exercise 3.6. Define a function `amazingCurve` with type signature

```
amazingCurve :: Int -> Int
```

that will double a student's score on an exam. However, if the new score after doubling is greater than 100, the function should output 100.

Exercise 3.7. What is the *type* of the expression `bagFee False` using the above definition of `bagFee`? What is the *value* of the expression `bagFee False` using the above definition of `bagFee`?

Exercise 3.8. "Give every function a type signature." In Haskell, it is good practice to give every function that you define in your program file a type signature. We have been doing this all along. Type signatures serve as a form of documentation to readers of your program (including yourself).

Add type signatures for each of the definitions in the code below.

```
circleRadius = 3.5

cot x = 1 / tan x

fe epsilon = epsilon * tan (epsilon * pi / 2)

fo epsilon = -epsilon * cot (epsilon * pi / 2)

g nu epsilon = sqrt (nu**2 - epsilon**2)
```

Chapter 4

Lists

4.1 List basics

A *list* in Haskell is an ordered collection of data, all with the same type. Here is an example of a list.

```
velocities :: [Double]
velocities = [0,-9.8,-19.6,-29.4]
```

The type `[Double]` indicates that `velocities` is a list of `Double`s. Square brackets in the type indicate a list. A list with type `[Double]` can have any number of items (including zero), but each item must have type `Double`. In the second line, we define `velocities` by enclosing its elements in square brackets separated by commas.

The empty list is denoted `[]`.

There is a list element operator `!!` that can be used to learn the value of individual elements of a list.

```
GHCi  :l Lists.lhs

GHCi  velocities !! 0
          ⇝  0.0
GHCi  velocities !! 1
          ⇝  -9.8
GHCi  velocities !! 3
          ⇝  -29.4
```

Note that the first element of a list is considered to be element number 0.

Lists of the same type can be concatenated with the `++` operator from Table 1.2.

GHCi
```
velocities ++ velocities
    ⇝  [0.0,-9.8,-19.6,-29.4,0.0,-9.8,-19.6,-29.4]
```

An *arithmetic sequence* is a list formed with two dots (`..`).

```
ns :: [Int]
ns = [0..10]
```

The list `ns` contains the integers from 0 to 10. I chose the name `ns` because it looks like the plural of the name `n`, which seems like a good name for an integer. It is a common style in Haskell programs to use names that end in `s` for lists, but it is by no means necessary.

If we type a list into GHCi,

GHCi
```
[0,2,5+3]
    ⇝  [0,2,8]
```

GHCi will evaluate each element and return the list of evaluated elements. If we give GHCi an arithmetic sequence,

GHCi
```
[0..10]
    ⇝  [0,1,2,3,4,5,6,7,8,9,10]
```

GHCi will expand the list for us.

A second form of arithmetic sequence allows an increment from one term to the next that is different from `1`.

GHCi
```
[-2,-1.5..1]
    ⇝  [-2.0,-1.5,-1.0,-0.5,0.0,0.5,1.0]
```

In this second form, one specifies the first, second, and last entries of the desired list. You can even do a decreasing list.

GHCi
```
[10,9.5..8]
    ⇝  [10.0,9.5,9.0,8.5,8.0]
```

A second way to form a new type from an existing type (recall the first way was function types, section 3.2) is to make a *list type*. Given any type `a` (`Int`, `Integer`, `Double`, etc.), there is a type `[a]` for lists with elements of type `a`.

You can, for example, make a list of functions. Recall the `square` function that we defined in chapter 2.

```haskell
square :: Double -> Double
square x = x**2
```

We can define the following list, where `cos` and `sin` are functions defined in the Haskell Prelude.

```haskell
funcs :: [Double -> Double]
funcs = [cos,square,sin]
```

Why would we want a list of functions? Later we will see a way to turn a list of functions into a plot of all of the functions on the same set of axes.

The Prelude provides a function `length` that returns the number of items in a list.

```
GHCi  :l Lists.lhs
```

The GHCi command `:l Lists.lhs` loads a *literate Haskell* file called `Lists.lhs`. Literate Haskell files are files that contain text (intended for people) as well as Haskell code (intended for people and computers). This chapter exists as a literate Haskell file called `Lists.lhs` that contains the code above defining `velocities`, `ns`, and `funcs`. After the file is loaded, we can use the `length` function to ask about the size of the lists.

```
GHCi  length velocities
          ⇝  4
GHCi  length ns
          ⇝  11
GHCi  length funcs
          ⇝  3
```

Table 4.1 shows some Prelude functions for working with lists. The function `head` returns the first element of a list. Some uses of `head` are given in Table 4.2. The function `head` can accept a list of type `[Double]`, a list of type `[Char]`, a list of type `[Int]`, or a list of type anything. Haskell allows the use of a *type variable* to indicate that a function can work with an arbitrary

Function		Type	Description
`head`	`::`	`[a] -> a`	return first item of list
`tail`	`::`	`[a] -> [a]`	return all but first item of list
`last`	`::`	`[a] -> a`	return last item of list
`init`	`::`	`[a] -> [a]`	return all but last item of list
`reverse`	`::`	`[a] -> [a]`	reverse order of list
`repeat`	`::`	`a -> [a]`	infinite list of a single item
`cycle`	`::`	`[a] -> [a]`	infinite list repeating given list

Table 4.1: Some Prelude functions for working with lists

type. The type of `head` is `[a] -> a`, in which `a` is a type variable that could be any type. The meaning of the type is that `head` will accept as input a list of type `a` (where `a` could be anything) and return an `a`. You can see a type variable if you ask GHCi for the type of the empty list.

GHCi
```
:t []
    ⇝  [] :: [a]
```

The `tail` function returns everything but the first element of a list. The function `last` returns the last element of a list. The function `init` returns everything except the last element. The online book *Learn You a Haskell for Great Good* (`http://learnyouahaskell.com/`) has cute a picture of a caterpillar that explains these list functions. The types of these functions are given in Table 4.1. Examples of their use are shown in Table 4.2.

Now that we have introduced lists, we can tell you that a string in Haskell is nothing but a list of characters. In other words, the type `String` is exactly the same as the type `[Char]`. Haskell provides some special syntax for strings, namely the ability to enclosed a sequence of characters in double quotes to form a `String`. This is obviously more pleasant than requiring an explicit list of characters, such as `['W','h','y','?']`. You can ask GHCi whether this is the same as `"Why?"`.

GHCi
```
['W','h','y','?'] == "Why?"
    ⇝  True
```

GHCi responds with `True`, indicating that it regards these two expressions as identical.

The identity of the types `String` and `[Char]` also means that a string can be used in any function that expects a list of something. For example,

Expression		evaluates to
`length ["Gal","Jo","Isaac","Mike"]`	⇝	`4`
`length [1, 2, 4, 8, 16]`	⇝	`5`
`head ["Gal","Jo","Isaac","Mike"]`	⇝	`"Gal"`
`head [1, 2, 4, 8, 16]`	⇝	`1`
`tail ["Gal","Jo","Isaac","Mike"]`	⇝	`["Jo","Isaac","Mike"]`
`tail [1, 2, 4, 8, 16]`	⇝	`[2,4,8,16]`
`init ["Gal","Jo","Isaac","Mike"]`	⇝	`["Gal","Jo","Isaac"]`
`init [1, 2, 4, 8, 16]`	⇝	`[1,2,4,8]`
`last ["Gal","Jo","Isaac","Mike"]`	⇝	`"Mike"`
`last [1, 2, 4, 8, 16]`	⇝	`16`

Table 4.2: Use of list functions.

we can use the function `length` on a string to tell us how many characters it has.

There will be times when you may wish to "bundle together" expressions of different types. For example, we may wish to form pairs composed of a person's name (a `String`) and age (an `Int`). A list is not the right structure to use for this job. All elements of a list must have the same type. In Chapter 8, we will learn about *tuples*, which are a good way to bundle together items of different types.

4.2 Infinite lists

Haskell is a lazy language, meaning that its does not always evaluate everything in the order you might expect, but waits to see if values are needed before doing any actual work. Haskell's laziness allows the possibility of infinite lists. Of course, Haskell never actually creates an infinite list, but you can think of the list as infinite because Haskell is willing to continue down the list as far as it needs to. The list `[1..]` is an example of an infinite list. If you ask GHCi to show you this list, it will go on indefinitely. You can type Control-C or something similar to stop the endless printing of numbers.

An infinite list can be convenient when you don't know in advance exactly how much of a list you will want or need. For example, we might want to compute a list of positions of a particle at 0.01 s time increments. We may not know in advance the length of time over which we want this information. Maybe we need to plot the values for the first 5 seconds, and then see if we need to go for a longer time interval. If we write our function so that it returns an infinite list of positions, the function will be simpler, because it doesn't need to know the total number of positions to calculate.

A good way to view the first several elements of an infinite list is with the `take` function. Try the following in GHCi.

GHCi
```
take 10 [3..]
    ⇝  [3,4,5,6,7,8,9,10,11,12]
```

GHCi should show you the first ten elements of the infinite list `[3..]`.

Two Prelude functions from Table 4.1 create infinite lists. The function `repeat` takes a single expression and returns an infinite list with the expression repeated an infinite number of times. By itself, this function doesn't seem very useful, but in combination with other functions we'll learn about later, it can be helpful.

The Prelude function `cycle` takes a (finite) list and returns the infinite list formed by cycling through the elements of the finite list over and over again. You can get an idea of what `cycle` does by asking GHCi to show you the first several elements of such a list, like the following.

GHCi
```
take 10 (cycle [4,7,8])
    ⇝  [4,7,8,4,7,8,4,7,8,4]
```

4.3 List constructors and pattern matching

There is a colon operator `:` (called *cons* for historical reasons) that attaches a single item of type `a` to a list with type `[a]`. For example, `3:[4,5]` is the same as `[3,4,5]`, and `3:[]` is the same as `[3]`.

Earlier we saw how we could use pattern matching to define a function that took a `Bool` as input. The idea was that a `Bool` can only be one of two things, so we'll just define explicitly how the function should behave for each of those two things. We can also use pattern matching to define a function that takes a list as input. The idea is that a list is *either* the empty list `[]` or the cons `x:xs` of an item `x` with a list `xs`. Every list is exactly one of these two mutually exclusive and exhaustive possibilities. In fact, internally Haskell regards lists as begin formed out of the two *constructors* `[]` and `:`. The list we think of as `[13,6,4]` is represented internally as `13:6:4:[]`

which means `13:(6:(4:[]))` when we allow for the right associativity of `:`. Let's give an example of defining a function on lists using pattern matching.

Example 4.1. Define a function `sndItem` that returns the second element of a list, or gives an error if the list has fewer than two elements. The idea is that `sndItem [8,6,7,5]` should return `6`. Here is our definition.

```
sndItem :: [a] -> a
sndItem []     = error "Empty list has no second element."
sndItem (x:xs) = if null xs
                 then error "1-item list has no 2nd item."
                 else head xs
```

This example used the `error` function, which has type `[Char] -> a`, meaning that it takes a string as input and can serve as any type. The `error` function halts execution and returns the given string as a message.

We can make an even nicer function definition by going one step further, using pattern matching on the `xs` list in `sndItem`.

Example 4.2. Define a function `sndItem'` that returns the second element of a list, or gives an error if the list has fewer than two elements. Try to write the function without an if-then-else construction.

```
sndItem' :: [a] -> a
sndItem' []      = error "Empty list has no second element."
sndItem' (x:[])  = error "1-item list has no 2nd item."
sndItem' (x:y:_) = y
```

Notice the underscore character in the last line. We could have written `(x:y:ys)` or `(x:y:xs')` in place of `(x:y:_)`. The underscore means that we can't be bothered to give the list a name because we have no intention of using it or referring to it again. For this function, we don't care whether the list of items after the second item is empty or not; we care so little about that list that we're not even going to give it a name.

4.4 Exercises

Exercise 4.1. Give an abbreviation for the following list using the double dot `..` notation. Use GHCi to check that your expression does the right thing.

```
ts :: [Double]
ts = [-2.0,-1.2,-0.4,0.4,1.2,2.0]
```

Exercise 4.2. Write a function `sndItem'' :: [a] -> a` that does the same thing as `sndItem`, but does not use pattern matching.

Exercise 4.3. What is the type of the following expression?

```
length "Hello, world!"
```

What is the value of the expression?

Exercise 4.4. Write a function with type `Int -> [Int]` and describe in words what it does.

Exercise 4.5. Write a function `null'` that does the same thing as the Prelude function `null`. Use the Prelude function `length` in your definition of `null'`, but do not use the function `null`.

Exercise 4.6. Write a function `last'` that does the same thing as the Prelude function `last`. Use the Prelude functions `head` and `reverse` in your definition of `last'`, but do not use the function `last`.

Exercise 4.7. Write a function `palindrome :: String -> Bool` that returns `True` if the input string is a palindrome (a word like "radar" that is spelled the same backwards as it is forwards), and `False` otherwise.

Exercise 4.8. What are the first five elements of the infinite list `[9,1..]`?

Exercise 4.9. Write a function `cycle'` that does the same thing as the Prelude function `cycle`. Use the Prelude functions `repeat` and `concat` in your definition of `cycle'`, but do not use the function `cycle`.

Exercise 4.10. Which of the following are valid Haskell expressions? If an expression is valid, give its type. If an expression is not valid, say what is wrong with it.

(a) `["hello",42]`

(b) `['h',"ello"]`

(c) `['a','b','c']`

(d) `length ['w','h','o']`

(e) `length "hello"`

(f) `reverse` (Hint: This is a valid Haskell expression, and has a well-defined type, even though GHCi cannot print the expression.)

Exercise 4.11. In an arithmetic sequence, if the specified last element does not occur in the sequence,

GHCi
```
[0,3..8]
    ⇝  [0,3,6]
```

GHCi
```
[0,3..8.0]
    ⇝  [0.0,3.0,6.0,9.0]
```

the result seems to depend on whether you are using whole numbers or not. Explore this and try to find a general rule for where an arithmetic sequence will end.

Chapter 5

Higher-order Functions

5.1 Functions with parameters

Consider the force of a linear spring with spring constant k. We usually write this as

$$F_{\text{spring}} = -kx$$

where the negative sign indicates that the force acts in the direction opposite the displacement.

Suppose we wish to write a Haskell function to give the force in Newtons produced by a spring with spring constant 5,500 N/m. We could write

```
springForce5500 :: Double -> Double
springForce5500 x = -5500 * x
```

This is a fine function, but it only handles the force produced by a spring with a spring constant of 5500 N/m. It would be nicer to have a function that could handle a spring with any spring constant. Consider the following function.

```
springForce :: Double -> Double -> Double
springForce k x = -k * x
```

The type for `springForce`, `Double -> Double -> Double` is equivalent to `Double -> (Double -> Double)` meaning that if we send the `springForce` function a Double (the spring constant), it will return to us a *function* with type Double -> Double. This latter function wants a Double as input (the position), and will give a Double as output (the force).

We can look at the types of these functions using GHCi's :type command.

```
GHCi   :l HigherOrder.lhs
```

```
GHCi   :t springForce
           ⇝  springForce :: Double -> Double -> Double
```

Next, let's look at the function springForce 2200.

```
GHCi   :t springForce 2200
           ⇝  springForce 2200 :: Double -> Double
```

The function springForce 2200 represents the force function (input: position, output: force) for a spring with spring constant 2200 N/m. It has the same type and plays the same role as the springForce5500 function above. It looks funny, because it is a function made up of two parts: the springForce part and the the 2200 part.

Finally, look at the type of springForce 2200 0.4.

```
GHCi   :t springForce 2200 0.4
           ⇝  springForce 2200 0.4 :: Double
```

This is not a function, but just a Double, representing the force exerted by a spring with spring constant 2200 N/m when extended by a distance of 0.4 m.

A function that takes another function as input or returns another function as a result is called a *higher-order function*. The function springForce is a higher-order function because it returns a function as its result. Higher-order functions give us a convenient way to define a function that takes one or more parameters (like the spring constant) in addition to its "actual" input (like the distance). Table 5.1 shows some higher-order functions from the Prelude that return a function as output.

Function		Type
`take`	`::`	`Int -> [a] -> [a]`
`drop`	`::`	`Int -> [a] -> [a]`
`replicate`	`::`	`Int -> a -> [a]`

Table 5.1: Some higher-order functions from the Prelude that produce a function as output

Consider the higher-order function `take`. The function `take` produces a list by taking a given number of elements from a given list. Table 5.2 shows some examples of its use.

Expression		evaluates to
`take 3 [9,7,5,3,17]`	$\rightsquigarrow$	`[9,7,5]`
`take 3 [3,2]`	$\rightsquigarrow$	`[3,2]`
`take 4 [1..]`	$\rightsquigarrow$	`[1,2,3,4]`
`take 4 [-10.0,-9.5..10]`	$\rightsquigarrow$	`[-10.0,-9.5,-9.0,-8.5]`

Table 5.2: Examples of the use of `take`

Let's look at the type of take.

```
take :: Int -> [a] -> [a]
```

According to the type of `take`, when given an `Int`, it should return a function with type `[a] -> [a]`. What function should `take` return? If we give the integer n to `take`, the returned function will accept a list as input and return a list of the first n elements of the input list.

There are two ways to think about the higher-order function `take` (and others like it that return a function as output), as shown in Table 5.3. We have already described the "1-input thinking", in which we read the type signature of `take` as expecting a single `Int` as input and producing an `[a] -> [a]` as output. An alternative way to think about the type signature `Int -> [a] -> [a]` is that what is expected is two inputs, the first of type `Int` and the second of type `[a]`, and that what is produced is an output of type `[a]`.

Way of thinking	input to `take`	output from `take`
1-input thinking	`Int`	`[a] -> [a]`
2-input thinking	`Int` and then `[a]`	`[a]`

Table 5.3: Two ways of thinking about the higher-order function `take`

Consider the function `drop`. The function `drop` produces a list by discarding a given number of elements from a given list. Table 5.4 shows some examples of its use.

Expression		evaluates to
`drop 3 [9,7,5,10,17]`	⇝	`[10,17]`
`drop 3 [4,2]`	⇝	`[]`
`drop 37 [-10.0,-9.5..10]`	⇝	`[8.5,9.0,9.5,10.0]`

Table 5.4: Examples of the use of `drop`

The function `replicate` produces a list by repeating one item a given number of times. Table 5.5 shows some examples of its use.

Expression		evaluates to
`replicate 2 False`	⇝	`[False,False]`
`replicate 3 "ho"`	⇝	`["ho","ho","ho"]`
`replicate 4 5`	⇝	`[5,5,5,5]`
`replicate 3 'x'`	⇝	`"xxx"`

Table 5.5: Examples of the use of `replicate`

Table 5.6 shows two ways of thinking about the higher-order function `replicate :: Int -> a -> [a]`. The "1-input" way of thinking, which is the way that the Haskell compiler uses, regards the input as an `Int` and the output as a function `a -> [a]`. The alternative "2-input" way of thinking regards the inputs as an `Int` and an `a` (where `a` is a type variable that stands for any type), and the output as `[a]`.

Way of thinking	input to `replicate`	output from `replicate`
1-input thinking	`Int`	`a -> [a]`
2-input thinking	`Int` and then `a`	`[a]`

Table 5.6: Two ways of thinking about the higher-order function `replicate`

In this section, we've focused on higher-order functions that return functions as results. Now let's take a look at a higher-order function that takes a function as input.

5.2 Numerical integration

A numerical integrator is a great example of a higher-order function because it takes a function as input. By numerical integration, we mean computing the value of a definite integral of some given function over some given limits. What we would like to be able to do is to give the computer a function f, give the computer limits a and b, and ask it to compute the number

$$\int_a^b f(x)\,dx.$$

First, we have to decide what algorithm to use to do the numerical integration. There are many to choose from, but we'll describe a simple and intuitively reasonable one, the trapezoidal rule.

In the trapezoidal rule, we approximate the area under a curve by the sum of the areas of a bunch of trapezoids, as shown in figure 5.1.

For the example in the figure, the area of the first trapezoid is

$$\frac{1}{2}[f(x) + f(x + \Delta x)]\Delta x.$$

The area of all four trapezoids in the figure is

$$\left(\frac{1}{2}f(x) + f(x + \Delta x) + f(x + 2\Delta x) + f(x + 3\Delta x) + \frac{1}{2}f(x + 4\Delta x)\right)\Delta x$$

The function `trapIntegrate` defined below does numerical integration.

Example 5.1. Write a definition for the function

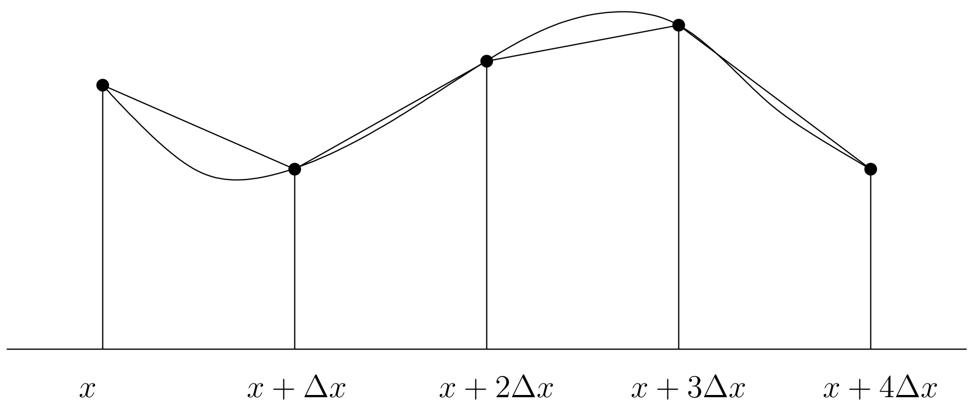


Figure 5.1: The trapezoidal rule.

```haskell
trapIntegrate :: Int
              -> (Double -> Double)
              -> Double
              -> Double
              -> Double
```

that takes a number of trapezoids, a function, and two limits as its arguments and gives back (an approximation to) the definite integral, using the trapezoidal rule. Test your integrator on the following integrals, and see how close you can get to the correct values.

$$\int_0^1 x^3\,dx = 0.25$$

$$\int_0^{10^{-6}} x^3\,dx = 2.5 \times 10^{-25}$$

$$\int_0^1 e^{-x^2}\,dx \approx 0.7468$$

Solution:

```
trapIntegrate :: Int
              -> (Double -> Double)
              -> Double
              -> Double
              -> Double
trapIntegrate n f a b
    = let dx          = (b - a) / fromIntegral n
          leftSides   = [a, a+dx .. b-dx]
          trapArea x = 0.5 * (f x + f (x+dx)) * dx
      in sum $ map trapArea leftSides
```

The `let` keyword introduces a local variable or function that can be used in the body after the `in` keyword.

Note how we use a single identifier (`f`) to name the function that the user of `trapIntegrate` passes in. Note also that we don't need to define the function `f`; what we are doing here is *naming* the function that the user of `trapIntegrate` is sending in.

GHCi
```
:l HigherOrder.lhs
```

GHCi
```
trapIntegrate 100 (\x -> x**3) 0 1
    ⇝  0.25002500000000044
```

5.3 Anonymous higher-order functions

In Section 2.3, we discussed anonymous functions as a way to describe a function without giving it a name. We can do the same thing for higher-order functions, describing them without giving them a name.

Let's return to the function `springForce` above. How could we write `springForce` without naming it? There are actually two ways to write this function as an anonymous function, corresponding to the 1-input thinking and 2-input thinking that we described in Section 5.1. In 1-input thinking, we regard the input to `springForce` as being a number (a `Double`), and the output as being a function `Double -> Double`. The anonymous function for 1-input thinking is shown in the first row of Table 5.7. We can see from the form of the anonymous function that it returns a function. In 2-input thinking, we regard the inputs to `springForce` as a `Double` for the spring constant and a second `Double` for the position, and the output as being simply a `Double`. The anonymous function for 2-input thinking is shown in the second row of Table 5.7. We can see from the form of the anonymous function that it returns a number. Either form is completely legitimate. In fact the two forms describe the same function.

Way of thinking	Anonymous function
1-input thinking	`\k -> \x -> -k*x`
2-input thinking	`\k x -> -k*x`

Table 5.7: Two ways of writing the `springForce` function as an anonymous function

5.4 Mapping over lists

Table 5.8 shows some higher-order Prelude functions that take other functions as input.

Function		Type
`map`	`::`	`(a -> b) -> [a] -> [b]`
`iterate`	`::`	`(a -> a) -> a -> [a]`

Table 5.8: Some higher-order functions from the Prelude that accept a function as input

The Prelude function `map` is a nice example of a higher-order function that takes another function as input. The function `map` will apply the function you give to every element of the list you give. Table 5.9 shows some examples of the use of `map`.

Expression		evaluates to
`map sqrt [1,4,9]`	$\rightsquigarrow$	`[1.0,2.0,3.0]`
`map length ["Four","score","and"]`	$\rightsquigarrow$	`[4,5,3]`
`map (logBase 2) [1,64,1024]`	$\rightsquigarrow$	`[0.0,6.0,10.0]`
`map reverse ["Four","score"]`	$\rightsquigarrow$	`["ruoF","erocs"]`

Table 5.9: Examples of the use of `map`

In the first example in Table 5.9, we say that the function `sqrt` gets “mapped” over the list, meaning that it gets applied to each element of the list.

The Prelude function `iterate` is another nice example of a higher-order function that takes a function as input. The function `iterate` produces an infinite list as follows. If `f :: a -> a` and `x :: a` then `iterate f a` produces the infinite list

```
[x, f x, f (f x), f (f (f x)), ...]
```

Table 5.10 shows some examples of the use of `iterate`.

Expression		evaluates to
`iterate (\n -> 2*n) 1`	⇝	`[1,2,4,8,...]`
`iterate (\n -> n*n) 1`	⇝	`[1,1,1,1,...]`
`iterate (\n -> n*n) 2`	⇝	`[2,4,16,256,...]`
`iterate (\v -> v - 9.8*0.1) 4`	⇝	`[4.0,3.02,2.04,1.06,...]`

Table 5.10: Examples of the use of `iterate`

5.5 Operators as higher-order functions

In Section 1.3, we introduced a number of infix operators in Table 1.2. Any infix operator can be converted into a higher-order function by enclosing it in parentheses. Table 5.11 shows examples of how infix operators may be written as higher-order functions.

Infix expression	equivalent prefix expression
`f . g`	`(.) f g`
`’A’:"moral"`	`(:) ’A’ "moral"`
`[3,9] ++ [6,7]`	`(++) [3,9] [6,7]`
`True && False`	`(&&) True False`
`p }} q`	`(}}) p q`
`log . sqrt $ 10`	`($) (log . sqrt) 10`

Table 5.11: An infix operator can be transformed into a (prefix) function by enclosing the operator in parentheses.

The types of the higher-order functions obtained from operators are shown in Table 5.12.

Function		Type
`(.)`	`::`	`(b -> c) -> (a -> b) -> a -> c`
`(:)`	`::`	`a -> [a] -> [a]`
`(++)`	`::`	`[a] -> [a] -> [a]`
`(&&)`	`::`	`Bool -> Bool -> Bool`
`(||)`	`::`	`Bool -> Bool -> Bool`
`($)`	`::`	`(a -> b) -> a -> b`

Table 5.12: Infix operators viewed as higher-order functions

5.6 Predicate-based higher-order functions

A *predicate* is a function with type `a -> Bool`, where `a` is any valid Haskell type. (For example, `a` could be a concrete type like `Int` or `Double -> Double`, a type variable like `a`, or a type that contains type variables like `[a]` or even `a -> [b]`.) A predicate expresses a property that an element of type `a` may or may not have. For example, the property of an integer being greater than or equal to seven is a predicate. We can define such a predicate in Haskell.

```
greaterThanOrEq7 :: Int -> Bool
greaterThanOrEq7 n = if n >= 7 then True else False
```

Table 5.13 shows a number of higher-order functions that take a predicate as the first argument.

Function		Type
`filter`	`::`	`(a -> Bool) -> [a] -> [a]`
`takeWhile`	`::`	`(a -> Bool) -> [a] -> [a]`
`dropWhile`	`::`	`(a -> Bool) -> [a] -> [a]`

Table 5.13: Some predicate-based higher-order functions from the Prelude

Let's examine the use of these functions. Suppose we define the following "less than 10" predicate.

```
lt10 :: Int -> Bool
lt10 n = n < 10
```

Table 5.14 shows examples of how to use the higher-order functions in Table 5.13.

Expression		evaluates to
`filter    lt10 [6,4,8,13,7]`	$\rightsquigarrow$	`[6,4,8,7]`
`takeWhile lt10 [6,4,8,13,7]`	$\rightsquigarrow$	`[6,4,8]`
`dropWhile lt10 [6,4,8,13,7]`	$\rightsquigarrow$	`[13,7]`
`any       lt10 [6,4,8,13,7]`	$\rightsquigarrow$	`True`
`all       lt10 [6,4,8,13,7]`	$\rightsquigarrow$	`False`

Table 5.14: Examples of the use of some predicate-based higher-order functions

5.7 Exercises

Exercise 5.1. Let us return to the example of throwing a rock straight upward. Perhaps we don't want to throw it upward at 30 m/s, but would like to be able to throw it upward with whatever initial velocity we choose. Write a function

```
yRock :: Double -> Double -> Double
```

that takes as input an initial velocity and returns as output a function that takes as input a time and returns as output a height. Also, write a function

```
vRock :: Double -> Double -> Double
```

that takes as input an initial velocity and returns as output a function that takes as input a time and returns as output a velocity.

Exercise 5.2. Give the type of `take 4`.

Exercise 5.3. The function `map` has type `(a -> b) -> [a] -> [b]`. This means that `map` is expecting a function with type `a -> b` as its first argument. The function `length` has type `[a] -> Int`. Can `length` be the first argument to `map`? Ask GHCi for the type of `map length`. Show how, starting from the types of `map` and `length`, you can figure out the type of `map length`.

Exercise 5.4. Write a function with type `Int -> String -> Bool` and describe in words what it does.

Exercise 5.5. Write a predicate with type `Int -> Bool` expressing the property "greater than or equal to seven" without using an if-then-else construction.

Exercise 5.6. Write a predicate expressing the property "has more than 6 elements" that takes a list as input. Include a type signature with your predicate definition.

Exercise 5.7. Table 5.5 gives examples of the use of the `replicate` function. In the first three examples, a list is created with the requested length of the requested item. In the last case, a string is created. This seems different. Explain what is going on here.

Exercise 5.8. Make a list of the first 1000 squares. Don't print the list; just print your definition. You could print the first 10 squares to see if your method is working.

Exercise 5.9. Use `iterate` to define a function `repeat'` that does the same thing as the Prelude function `repeat`.

Exercise 5.10. Use `take` and `repeat` to define a function `replicate'` that does the same thing as the Prelude function `replicate`.

Chapter 6

Quick Plotting

There are times when you want to make a quick plot to see what a function looks like. Here is an example of how to do this using GHCi.

GHCi
```
:m Graphics.Gnuplot.Simple
```

GHCi
```
plotFunc [] [0,0.1..10] cos
   ⇝
```

The first command loads a graphics module that can make graphs. (The GHCi command `:m` is short for `:module`, which loads a module of predefined functions.) The second command plots the function `cos` from 0 to 10 in increments of 0.1. This is carried out by the `plotFunc` function, which is one of the functions loaded in the `Graphics.Gnuplot.Simple` module. The `plotFunc` function takes a list of attributes (in this case, the empty list `[]`), a list of values at which to compute the function (in this case, `[0,0.1..10]`, which is a list of 101 numbers from 0 to 10 in increments of 0.1), and a function to plot (in this case, `cos`).

A hundred points is usually enough points to get a nice smooth graph. If it's not smooth enough for you, you could use 500 points or more. If you use only 4 points, you won't get a smooth graph (try it and see what happens). You can press the key `q` to make the plot window disappear.

In a later section, we'll learn how to make a nice plot with a title and axis labels for a presentation or an assignment.

If you wish to plot a function that is defined in a program file, you have a few choices.

Method 1: Put only the function to be plotted in the program file.

Suppose, for example, we want to plot the `square` function that we defined in our `first.hs` program file from $x = -3$ to $x = 3$. We could issue the following sequence of commands.

```
ghci> :m Graphics.Gnuplot.Simple
ghci> :l first.hs
ghci> plotFunc [] [-3,-2.99..3] square
```

Method 2: Use the program file to define the function to be plotted and to load the graphing module.

If we know that a program file contains a function or functions that we will want to plot, we can import the `Graphics.Gnuplot.Simple` module in the program file so that we don't have to do it at the GHCi command line. Instead of typing the `:m Graphics.Gnuplot.Simple` line into GHCi, we can put the following line at the top of our program file.

```
import Graphics.Gnuplot.Simple
```

Now in GHCi we do

```
ghci> :l first.hs
ghci> plotFunc [] [-3,-2.99..3] square
```

Method 3: Use the program file to load the graphing module, to define the function to be plotted, and to define the plot.

If we know in advance what plot we want, we can include the plotting commands in the program file itself. In our program file, we can define the plot

```
plot1 = plotFunc [] [-3,-2.99..3] square
```

Now to make our plot, we only need to issue the following commands in GHCi.

```
ghci> :l first.hs
ghci> plot1
```

6.1 Exercises

Exercise 6.1. Make a plot of $\sin(x)$ from $x = -10$ to $x = 10$.

Exercise 6.2. Make a plot of your `yRock30` function from $t = 0$ to $t = 6$ s.

Exercise 6.3. Make a plot of your `yRock 20` function from $t = 0$ to $t = 4$ s. You will need to enclose `yRock 20` in parentheses when you use it as an argument to `plotFunc`.

Chapter 7

Type Classes

7.1 Introduction

Haskell is an extremely powerful programming language, in the sense that it gives an amazing amount of expressiveness and control to the programmer who has mastered the language. As with most powerful tools, this extreme power comes at a cost of making the language harder to learn and more confusing for the beginner. In this chapter, we discuss a feature of the Haskell language, namely the notion of a *type class*, that is quite powerful, but which can be confusing to someone learning the language.

Let us begin by asking GHCi about the type of the number `4`.

GHCi
```
:t 4
    ⇝  4 :: Num p => p
```

The number `4` is an example of a *literal* expression in Haskell. This is in distinction from an expression composed of one or more names, or *identifiers*, such as `e`, `square`, or `yRock`. It would be entirely reasonable to expect the type of the number `4` to be `Int` or `Integer`. But the designers of the Haskell language wanted a number like `4` to be able to be an `Int`, an `Integer`, a `Double`, or even a few other types, depending on the programmer's needs. For this reason (and other, more compelling reasons), they invented the idea of *type classes*. A type class is like a club that a type can belong to. When you ask GHCi for the type of the number `4`, it says the following.

```
4 :: Num a => a
```

This should be read, "for every type `a` that is an instance of the class `Num`, `4` has type `a`" or "`4` has type `a` as long as `a` is in type class `Num`". This says that the number `4` can have any type `a` as long as `a` belongs to the type class `Num` (short for number). The letter `a` is a *type variable* in this type signature. It can stand for any type. The stuff to the left of `=>` are type class constraints. In the type signature above, there is one type class constraint, `Num a`. This says that `a` must belong to type class `Num`. As with concrete types (such as `Int` and `Double -> Double`), we use blue to indicate type variables. We use green to indicate type classes.

The types `Int`, `Integer`, and `Double` are all *instances* (which is to say members) of the type class `Num`. So, the line `4 :: Num a => a` can be read "as long as `a` is an instance of type class `Num`, the item `4` has type `a`."

Basically, GHCi hasn't committed to a particular type for the number `4` yet. But this noncommittal attitude about the type of `4` can't go on forever. At some point, the Haskell compiler will demand that every item have a well-defined type. The lack of a well-defined type for something can be a source of trouble. GHCi has some type-defaulting rules to make our life easier. For example, if you put the line

```
x = 4
```

into a program file (say `typetest.hs`), giving `x` no type signature, load it into GHCi, and then ask for the type of `x`,

```
GHCi  :l typetest.hs
GHCi  :t x
         ⇝  x :: Integer
```

GHCi will tell you that `x` has type `Integer`. So, here GHCi has committed to a particular type without our specifying the type.

There are other situations, and you will come across them soon enough, where GHCi feels unable to assign a type, and you will need to help it out by adding type signatures to your code.

7.2 Type classes from the Prelude

Table 7.1 shows (in the left column) a number of type classes provided by the Prelude. Also shown in this table are which of the basic types are instances of each of the type classes.

Type class	Bool	Char	Int	Integer	Float	Double
Eq	✓	✓	✓	✓	✓	✓
Ord	✓	✓	✓	✓	✓	✓
Show	✓	✓	✓	✓	✓	✓
Num			✓	✓	✓	✓
Integral			✓	✓		
Fractional					✓	✓
Floating					✓	✓

Table 7.1: Table showing which basic types are instances of various type classes

7.2.1 Type class Eq

Type class `Eq` is for types that have a notion of equality (in other words, types for which the operators `==` and `/=` are defined). You can see in Table 7.1 that all of the six basic types `Bool`, `Char`, `Int`, `Integer`, `Float`, and `Double` are instances of `Eq`. The type of the function `(==)` is

```
GHCi  :t (==)
        ⇝  (==) :: Eq a => a -> a -> Bool
```

which means that the operator `==` can be used between any two expressions of the same type `a`, as long as `a` is an instance of `Eq`. What sort of type would not be an instance of `Eq`? Generally function types are not instances of `Eq`. For example, the type `Double -> Double` is not an instance of `Eq`. The reason is that it is usually difficult or impossible to check whether two functions are equal.

From the perspective of computational physics, it is a bad idea that `Float` and `Double` are instances of `Eq`. Because these two types are used for approximate calculation, you should never test `Float`s and `Double`s for equality. From the perspective of the computer, these types are each represented by a finite number of bits, and the computer will happily can check whether each bit of one `Double` is the same as the corresponding bit of another `Double`. But as we saw in section 1.6, the bits of `sqrt 5 ^ 2` are not the same as the bits of `5`. They are very close, but not the same. The take home message for computational physics is to avoid using `==` for approximate types like `Double`.

7.2.2 Type class `Ord`

Type class `Ord` is for types that have a notion of order (in other words, types for which the operators `<`, `<=`, `>`, and `>=`, are defined). A type must first be an instance of `Eq` before it may be an instance of `Ord`. The type of the function `(<)` is

```
GHCi  :t (<)
        ⇝  (<) :: Ord a => a -> a -> Bool
```

which means that the operator `<` can be used between any two expressions of the same type `a`, as long as `a` is an instance of `Ord`.

7.2.3 Type class `Show`

Type class `Show` is for types whose elements can be shown on the screen. Function types are not typically instances of `Show`. If I type the name of a function at the GHCi prompt,

```
GHCi  sqrt
        ⇝  <interactive>:6:1: error:      No instance for (Show (Double
```

```
Prelude> sqrt
```

I get an error like the following.

```
<interactive>:2:1:
    No instance for (Show (a0 -> a0)) arising from a use of print
    In a stmt of an interactive GHCi command: print it
```

This error complains that there is no `Show` instance for `sqrt`. GHCi knows how to apply the `sqrt` function to numbers and show you the result, but it does not know how to show you the `sqrt` function itself. Note that `sqrt` is a perfectly acceptable Haskell expression, with a well-defined type, even though it cannot be shown.

7.2.4 Type class `Num`

Type class `Num` is for numeric types. You can see in Table 7.1 that the types `Int`, `Integer`, `Float`, and `Double` are instances of `Num`, while `Bool` and `Char` are not. The type of the function `(+)` is

```
(+) :: Num a => a -> a -> a
```

meaning that the operator `+` can be used between any two expressions of the same type `a`, as long as `a` is an instance of `Num`, and the result will be an expression of type `a`.

7.2.5 Type class `Integral`

Type class `Integral` is for types that behave like integers. A type must first be an instance of `Num` before it may be an instance of `Integral`. You can see in Table 7.1 that the types `Int` and `Integer` are instances of `Integral`, while `Float` and `Double` are not. The type of the function `rem`, which finds the remainder of one integer divided by another, is

```
rem :: Integral a => a -> a -> a
```

meaning that the function `rem` can be used between any two expressions of the same type `a`, as long as `a` is an instance of `Integral`, and the result will be an expression of type `a`.

7.2.6 Type class `Fractional`

Type class `Fractional` is for numeric types that support division. A type must first be an instance of `Num` before it may be an instance of `Fractional`. You can see in Table 7.1 that the types `Float` and `Double` are instances of `Fractional`, while `Int` and `Integer` are not. The type of the function `(/)` is

```
(/) :: Fractional a => a -> a -> a
```

meaning that the operator `/` can be used between any two expressions of the same type `a`, as long as `a` is an instance of `Fractional`, and the result will be an expression of type `a`.

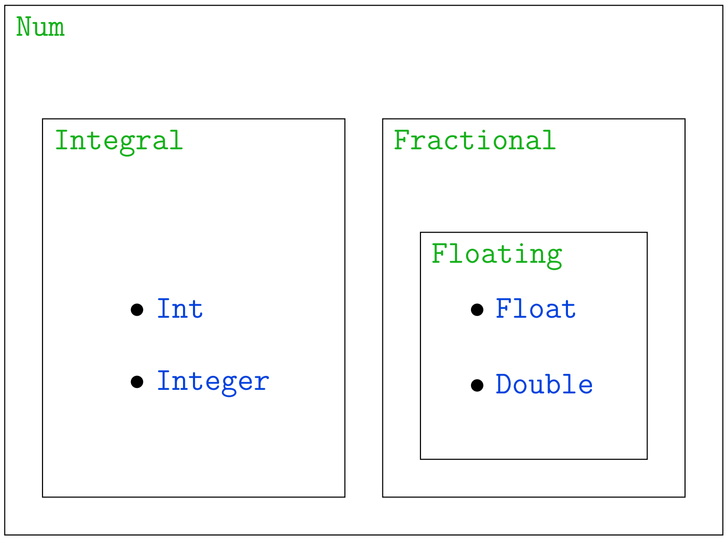


Figure 7.1: Relationship among the numeric type classes `Num`, `Integral`, `Fractional`, and `Floating`. Types `Int` and `Integer` are instances of type classes `Integral` and `Num`. Types `Float` and `Double` are instances of type classes `Floating`, `Fractional`, and `Num`.

7.2.7 Type class `Floating`

Type class `Floating` is for numeric types that are stored by the computer as "floating point" numbers, that is, inexact approximations. A type must first be an instance of `Fractional` before it may be an instance of `Floating`. You can see in Table 7.1 that the types `Float` and `Double` are instances of `Floating`, while `Int` and `Integer` are not. The type of the function `cos` is

GHCi
```
:t cos
    ⇝  cos :: Floating a => a -> a
```

meaning that the function `cos` can be used on any expression of the type `a`, as long as `a` is an instance of `Floating`, and the result will be an expression of type `a`.

Figure 7.1 shows the relationship among the numeric type classes we have just discussed.

7.3 Prelude functions with type class constraints

Table 7.2 shows some Prelude functions with type class constraints.

Function	Type
`div`, `mod`, `quot`, `rem`	`Integral a => a -> a -> a`
`(==)`,`(/=)`	`Eq a => a -> a -> Bool`
`(<)`,`(>)`,`(<=)`,`(>=)`	`Ord a => a -> a -> Bool`
`(+)`,`(-)`,`(*)`	`Num a => a -> a -> a`
`(/)`	`Fractional a => a -> a -> a`
`abs`	`Num a => a -> a`
`sin`, `exp`	`Floating a => a -> a`
`(^)`	`(Integral b, Num a) => a -> b -> a`
`(^^)`	`(Fractional a, Integral b) => a -> b -> a`
`(**)`	`Floating a => a -> a -> a`

Table 7.2: Some Prelude functions with type class constraints

7.4 Sections

An infix operator expects an argument on its left and an argument on its right. If only one of these two arguments were given, the resulting expression could be thought of as a function waiting for the other argument. Haskell allows us to make such functions by enclosing in parentheses an operator and one of its arguments. Examples are shown in Table 7.3. A function formed by enclosing an operator and one argument in parentheses is called a *section*.

Function	Type
`(+1)`	`Num a => a -> a`
`(2*)`	`Num a => a -> a`
`(. length)`	`(Int -> c) -> [a] -> c`
`('A':)`	`[Char] -> [Char]`
`(:"end")`	`Char -> [Char]`
`("I won't " ++)`	`[Char] -> [Char]`
`($ True)`	`(Bool -> b) -> b`

Table 7.3: Examples of the use of sections

For example, `(+1)` (which could also be written `(1+)`) is a function that adds one to its argument, and `(2*)` (which could also be written `(*2)`) is a function that doubles its argument.

7.5 Higher-order functions as operators

You can make a higher-order function into an infix operator by enclosing it in backquotes. For example, consider the Prelude function `elem`.

GHCi
```
:t elem
   ⇝  elem :: (Foldable t, Eq a) => a -> t a -> Bool
```

```
elem :: Eq a => a -> [a] -> Bool
```

The function `elem` takes an element of type `a`, a list of `a`s, and tells whether the element is in the list. The `Eq a` type class constraint exists because type `a` needs to have a sense of equality for this to work. Table 7.4 shows examples of the use of `elem`.

Expression	evaluates to
`elem 7 [7,14,21]`	`True`
`elem 8 [7,14,21]`	`False`
`elem 14 [7,14,21]`	`True`
``7 `elem` [7,14,21]``	`True`
``8 `elem` [7,14,21]``	`False`
``14 `elem` [7,14,21]``	`True`

Table 7.4: Examples of the use of the `elem` function

Of particular interest are the last three entries in the table, in which the function has been enclosed in backquotes and used as an infix operator.

7.6 Example of type classes and plotting

As an example, create a new program file called typeTrouble.hs with the following code.

```
import Graphics.Gnuplot.Simple

plot1 = plotFunc [] [0,0.01..10] cos
```

When I try to load this file into GHCi, I get the following horrible-looking error message.

```
typeTrouble.hs:3:8:
    Ambiguous type variable ‘t’ in the constraints:
      ‘Graphics.Gnuplot.Value.Tuple.C t’
        arising from a use of ‘plotFunc’ at typeTrouble.hs:3:8-35
      ‘Graphics.Gnuplot.Value.Atom.C t’
        arising from a use of ‘plotFunc’ at typeTrouble.hs:3:8-35
      ‘Floating t’ arising from a use of ‘cos’ at typeTrouble.hs:3:33-35
      ‘Enum t’
        arising from the arithmetic sequence ‘0, 1.0e-2 .. 10’
                     at typeTrouble.hs:3:20-31
    Probable fix: add a type signature that fixes these type variable(s)
```

First of all, don't panic. This error message contains much more information than we need to solve the problem. The most useful parts of the message are the first and last lines. The first line tells where the problem is in the code (line 3, column 8). At line 3, column 8 of our code is the function `plotFunc`. Let's look at the type of `plotFunc`.

```
ghci> :t plotFunc
```

Hmmm. Life just got worse. I get an error like this.

```
<interactive>:1:0: Not in scope: ‘plotFunc’
```

Now, this latter error is an easy one. “Not in scope” means that GHCi claims no knowledge of this function. That makes sense, actually, because it’s not included in the Prelude (the collection of built-in functions that are loaded automatically when we start up GHCi), and GHCi refused to load our `typeTrouble.hs` file because it had a problem with it. So, at the moment, it has no knowledge of `plotFunc`. Function `plotFunc` is defined in the module `Graphics.Gnuplot.Simple`. We can get access to `plotFunc` by loading the plotting module manually, like we first did to make a quick plot.

```
ghci> :m Graphics.Gnuplot.Simple
```

Now, let’s ask again for the type of `plotFunc`

```
ghci> :t plotFunc
```

Here’s what I get.

```
plotFunc
  :: (Graphics.Gnuplot.Value.Atom.C a,
      Graphics.Gnuplot.Value.Tuple.C a) =>
     [Attribute] -> [a] -> (a -> a) -> IO ()
```

There are a couple of type class constraints to the left of the `=>`. I don’t know what those type classes are, but as long as `a` (a type variable) belongs to those two type classes, the type of `plotFunc` is

```
[Attribute] -> [a] -> (a -> a) -> IO ()
```

In other words, `plotFunc` needs a list of `Attributes` (we have given an empty list in our examples so far), a list of `as`, and a function that takes an `a` as input and gives back an `a` as output. If we give `plotFunc` all this stuff, it will give us back an `IO ()`, which is a way of saying that it will actually *do* something for us (make a plot). Now, we would be perfectly happy if the type of `plotFunc` was

```
[Attribute] -> [Double] -> (Double -> Double) -> IO ()
```

without those crazy type class constraints.

Let’s return to the horrible-looking error message and focus on the last line.

```
Probable fix: add a type signature that fixes these type variable(s)
```

This tells us that the Haskell compiler would like more help figuring out the types of things. In particular, it can't figure out the types of `[0,0.01..10]` and `cos`. Let's ask GHCi about the types of these two.

```
ghci> :t [0,0.01..10]
```

I get

```
[0,0.01..10] :: (Fractional t, Enum t) => [t]
```

containing more type class gobbledygook. For

```
ghci> :t cos
```

I get

```
cos :: (Floating a) => a -> a
```

which also is type-class-constraint-laden.

One solution to the problem is to give the list `[0,0.01..10]` a name and a type signature. Make a program file `typeTrouble2.hs` with the following lines.

```
import Graphics.Gnuplot.Simple

xRange :: [Double]
xRange = [0,0.01..10]

plot2 = plotFunc [] xRange cos
```

This program file should load fine and give you a nice plot when you type `plot2`. Try it and see.

Another solution is to specify the type of the list `[0,0.01..10]` on the line where it's used. Make a program file `typeTrouble3.hs` with the following lines.

```
import Graphics.Gnuplot.Simple

plot3 = plotFunc [] ([0,0.01..10] :: [Double]) cos
```

Yet another solution is to tell the compiler that the final element of the list, `10`, has type `Double`. This implies that all of the elements in the list have type `Double`.

```
import Graphics.Gnuplot.Simple

plot4 = plotFunc [] [0,0.01..10 :: Double] cos
```

The moral of the story is that you should include type signatures for all of the functions you define, and you should be prepared to add more type signatures if the compiler complains.

7.7 Exercises

Exercise 7.1. Is it possible for a type to belong to more than one type class? If so, give an example. If not, why not?

Exercise 7.2. We said in this chapter that function types are typically not instances of `Eq`, because it is too hard to check whether two functions are equal.

(a) What does it mean mathematically for two functions to be equal?

(b) Why is it usually very hard or impossible for the computer to check if two functions are equal?

(c) Give a specific example of a function type that would be easy to check for equality.

Exercise 7.3. The function `(*2)` is the same as the function `(2*)`. Is the function `(/2)` the same as the function `(2/)`? Explain what these functions do.

Exercise 7.4. In section 2.1, we defined a function `square`. Now that we know that Haskell has sections, we can see that we didn't need to defined `square`. Show how to use a section to write the function that squares its argument.

Exercise 7.5. You can get information from GHCi about a type or a type class by using the GHCi command `:info` (`:i` for short), followed by the name of the type or type class you want information about. If you ask for information about a type, GHCi will tell you the type classes of which your type is an instance (the line `instance Num Double`, for example, means that the type `Double` is an instance of the type class `Num`). If you ask for information about a type class, GHCi will tell you the types which are instances of your type class.

(a) We showed in Table 7.1 the the type `Integer` was an instance of type classes `Eq`, `Ord`, `Show`, `Num`, and `Integral`. There are a few more type classes that we did not discuss of which `Integer` is also an instance. Find these.

(b) Type class `Enum` is for types that can be enumerated, or listed. Which Prelude types are instances of `Enum`?

Exercise 7.6. Find the types of the following Prelude Haskell expressions (some are functions and some are not).

1. `42`
2. `42.0`
3. `42.5`
4. `pi`
5. `[3,1,4]`
6. `[3,3.5,4]`
7. `[3,3.1,pi]`
8. `(==)`
9. `(/=)`
10. `(<)`
11. `(<=)`
12. `(+)`
13. `(-)`

14. `(*)`

15. `(/)`

16. `(^)`

17. `(**)`

18. `8/4`

19. `sqrt`

20. `cos`

21. `show`

22. `(2/)`

Exercise 7.7. If $8/4 = 2$, and `2 :: Num a => a` (2 has type `a` for every type `a` in type class `Num`) then why does `8/4 :: Fractional a => a`?

Chapter 8

Tuples

Haskell provides several ways of constructing new types from old. Given types `a` and `b`, we have already seen that there is a function type `a -> b`. Tuples are an even simpler way to construct new types from old. Given types `a` and `b`, there is a pair type `(a,b)` that serves as the type for ordered pairs in which the first item in the pair has type `a` and the second has type `b`. Given a further type `c`, there is a triple type `(a,b,c)` for ordered triples. Similarly, there are tuple types constructed from 4, 5, or more types. The types from which a tuple is constructed could be different, but they need not be.

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

8.1 Pairs

The simplest tuple is the pair. A pair is a combination of two values that already have types.

For example, here is a pair composed of a `String` to describe a person's name, and an `Int` to represent the person's score on an exam.

```
nameScore :: (String,Int)
nameScore = ("Albert Einstein", 79)
```

You see that the first component of a pair needs to have a well-defined type, and the second component of a pair also needs a type, but they need not be the same type.

Let's write a function `pythag1` that computes the hypotenuse of a right triangle from the lengths of the two sides. We'll pass the two side lengths to the function using a pair. Here is one way to write this function.

```
pythag1 :: (Double,Double) -> Double
pythag1 (a,b) = sqrt (a**2 + b**2)
```

The type signature above shows us that `pythag1` expects a pair of two `Double`s as input, and produces a `Double` as output. The fact that in the second line we call the input `(a,b)` (rather than a simple variable like `x`) means that this definition uses pattern matching. This is similar to the pattern matching we saw earlier for `Bool` and for lists. Pattern matching for tuples is simple because there is only one pattern. (Recall that `Bool` has two patterns, `True` and `False`, and that lists have two patterns, the emply list `[]`, and the cons of an element and a list `x:xs`.)

There are a couple of Prelude functions to deal with pairs. The `fst` function takes a pair as input and returns the first component of the pair as output. The `snd` function takes a pair as input and returns the second component of the pair as output. We can test this behavior in GHCi.

```
ghci> fst ("Albert Einstein", 79)
```

These two helper functions allow us to write an alternative version of `pythag1` that does the same thing.

```
pythag1' :: (Double,Double) -> Double
pythag1' pr = sqrt ((fst pr)**2 + (snd pr)**2)
```

The function `pythag1'` takes a simple variable as input; hence this function does *not* use pattern matching.

8.2 Functions of two variables

All Haskell functions have one input and one output. How then could we write a function of two variables, like

$$f(x,y) = \sqrt{x^2 + y^2}?$$

There are two ways to do this. One way is to use a tuple (a pair) as the input.

```
f :: (Double,Double) -> Double
f (x,y) = sqrt (x**2 + y**2)
```

Haskell regards a tuple as one thing, so Haskell thinks that the function `f` has one input (a `(Double,Double)`) and one output (a `Double`). Let's call this the *tuple form* of a function of two variables.

A second way to make a function of two variables is to use a higher-order function, as described in Section 5.1. In this way, we would encode the function

$$g(x, y) = \sqrt{x^2 + y^2}$$

as

```
g :: Double -> Double -> Double
g x y = sqrt (x**2 + y**2)
```

In this case, Haskell thinks that the one input to `g` is a `Double`, and that the one output of `g` is a `Double -> Double` (that is, a function from `Double` to `Double`). But you can think of `g` as a function of two variables, that accepts two `Double`s and returns a `Double`. Let's call this the *curried form* of a function of two variables.

These two ways of encoding a function of two variables (the tuple form and the curried form) are mutually exclusive. You need to pick one or the other for a particular function; you can't use both. Notice that the tuple form requires the use of parentheses and a comma around the two arguments. That's because you need to have a tuple as input! Notice that the curried form has no parentheses and no comma. It's not that the comma is optional; it must not be present.

Sometimes you might use one form, and realize later on that you wish you had used the other form. Haskell provides two functions to let you convert from one form to the other and back. To convert from tuple form to curried form, Haskell provides the function `curry`. We could write `curry f` and this would be exactly the same function as the `g` that we defined above. We can also write `uncurry g`, and this is the same as the function `f`. It does not make sense to write `curry g` or `uncurry f`; these constructions will produce type errors when the compiler tries to read them.

Take a look at the types for `curry` and `uncurry`. They might look like nonsense the first time you see them, but it's worth spending time to understand why they have the types they have. See if you can explain to yourself why they have the types they do.

8.3 Triples

In addition to pairs, you can make triples, or tuples with even more components. However, the functions `fst` and `snd` work only with pairs. To access elements of triples and larger tuples, the standard method is to use pattern matching. For example, functions that pick out the components of triples can be defined as follows.

```
fst3 :: (a,b,c) -> a
fst3 (x,y,z) = x

snd3 :: (a,b,c) -> b
snd3 (_,y,_) = y

thd3 :: (a,b,c) -> c
thd3 (_x,_y,z) = z
```

The definitions of `fst3`, `snd3`, and `thd3` use pattern matching to assign names to the items in the triple. These names can then be used on the right-hand side of the definition to indicate the value that we want the function to return. In the function `fst3`, the values `y` and `z` are not used. Because they are not used, it is in some sense superfluous to give them names.

In the definition of `snd3`, the underscore `_` is used as a place holder to represent a quantity that doesn't get used in the expression that follows. In the definition of `snd3`, we use underscores in the first and third slots of the triple. The point is that it is superfluous to give these items names since the names are not used in the definition.

In the definition of `thd3`, we show an alternate use of underscores.

Compiler warning.

8.4 Comparison of lists and tuples

A tuple is different from a list in that every element of a list must have the same type. On the other hand, the type of a tuple says exactly how many elements the tuple has. If an expression has type `[Int]`, for example, it is a list of zero, one, two, or more `Int`s. If an expression has type `(String,Int)`, it is a pair (2-tuple) consisting of exactly one `String` and exactly one `Int`. If you want to combine exactly two things or exactly three things, then a tuple is what you want. Beyond three items, tuples rapidly become unwieldy. Lists, on the other hand, are often very long. A list can happily contain thousands of elements.

8.5 `Maybe` types

The idea of `Maybe` types is independent of the idea of tuples. The only reason for introducing it here in this chapter on tuples is that I want to use it in the next section.

We saw in Chapter 4 on lists that for any type `a` there is another type `[a]` consisting of lists of elements that each have type `a`. Such a list may have zero, one, two, or more elements of type `a`.

In an analogous way, for any type `a` there is another type `Maybe a` consisting of zero or one element of type `a`. To motivate this new data type, imagine that you are writing a function `findFirst` that will search through a list for the first element that meets some criterion. We might wish such a function to have type `[b] -> b`.

```
findFirst :: [b] -> b
```

The type `[b] -> b` indicates our intent to have the function `findFirst` accept a list of elements of type `b` as input and provide a single element of type `b` as output. But what if the list contains no element that meets our criterion? In that case there is a problem because the function `findFirst` has no way to come up with an element of type `b`, but the type `[b] -> b` *demands* that the function return (produce) an element of type `b`. One possibly is for `findFirst` give an `error` if no suitable element is found, but this is an extreme measure, and will halt the program, so that no later recovery is possible. A better solution is to use a different type signature.

```
findFirst' :: [b] -> Maybe b
```

If `findFirst'` finds an element `x :: b` that meets the criterion, it will return `Just x`. If it finds no element of type `b` that meets the criterion, `findFirst'` will return `Nothing`.

Type type `Maybe a` has two patterns. (Recall that `Bool` has two patterns, lists have two patterns, and tuples have one pattern.) An element of `Maybe a` is either `Nothing` or `Just x` for some `x :: a`. The element `Nothing` is the way of specifying zero elements of type `a`, and the element `Just x` is the way of specifying one element of type `a` (namely `x`). Table 8.1 shows some expressions involving `Maybe` and their types. Table 8.2 shows a comparison of expressions having `Maybe` types with expressions having the underlying type.

Expression	Type
`Nothing`	`Maybe a`
`Just "me"`	`Maybe String`
`Just 'X'`	`Maybe Char`
`Just False`	`Maybe Bool`
`Just 4`	`Num a => Maybe a`

Table 8.1: Some expressions involving `Maybe` and their types.

Finally, we should point out that `Maybe Int` is a type, `Maybe Bool` is a type, and `Maybe Double` is a type, but `Maybe` itself is not a type. `Maybe` is called a *type constructor*. A type constructor is an object that takes one type as input and produces another type as output. We think of `Maybe` as taking the type `Int` as input and producing the type `Maybe Int` as output. In other words, `Maybe` is a function at the type level. In order to keep track of this complexity, Haskell assigns a *kind* to each type and type constuctor. A type, such as `Double` has kind `*`. GHCi has the command `:kind` (or `:k` for short) to ask about the kind of something.

GHCi

```
:k Double
    ⇝  Double :: *
```

A type constructor, such as `Maybe`, has kind `* -> *`.

Type	Expressions with this type
`Bool`	`False`, `True`
`Maybe Bool`	`Just False`, `Just True`, `Nothing`
`Char`	`'h'`, `'7'`
`Maybe Char`	`Just 'h'`, `Just '7'`, `Nothing`
`String`	`"Monday"`, `"Tuesday"`
`Maybe String`	`Just "Monday"`, `Just "Tuesday"`, `Nothing`
`Int`	`3`, `7`, `-13`
`Maybe Int`	`Just 3`, `Just 7`, `Just (-13)`, `Nothing`

Table 8.2: Comparison of expressions having `Maybe` types with expressions having the underlying type.

GHCi
```
:k Maybe
    ⇝  Maybe :: * -> *
```

Once we apply `Maybe` to `Double`, the resulting `Maybe Double` is once again a type, with kind `*`.

GHCi
```
:k Maybe Double
    ⇝  Maybe Double :: *
```

Types have kind `*`, type constructors have kind `* -> *`, and there are objects with more complicated kinds as well.

While we are talking about kinds, it is interesting to note that you can ask GHCi for the kind of a type class.

GHCi
```
:k Num
    ⇝  Num :: * -> Constraint
```

This kind means that, when provided with a type, the type class `Num` produces a constraint.

8.6 Lists of pairs

Just as we can form lists of lists, we can make pairs of pairs, lists of pairs, pairs of lists, and more complicated things. Probably the list of pairs is the most useful of these (although lists of lists are very useful), for reasons we will see below.

The Prelude function

```
zip :: [a] -> [b] -> [(a, b)]
```

takes two lists and forms a list of pairs. The function `zip` pairs the first elements of the two lists, the second elements of the two lists, and so on. Table 8.3 shows examples of the use of `zip`.

The Prelude function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

is a high-power relative of `zip` that actually does something with the pairs of elements that `zip` generates. The first argument to `zipWith` is a higher-order function that describes what to do with an element of type `a` (from the first list) and an element of type `b` (from the second list). The second argument to `zipWith` is the first list, and the third argument to `zipWith` is the second list. Table 8.3 shows examples of the use of `zipWith`.

Expression		evaluates to
`zip [1,2,3] [4,5,6]`	⇝	`[(1,4),(2,5),(3,6)]`
`zip [5..7] "who"`	⇝	`[(5,'w'),(6,'h'),(7,'o')]`
`zipWith (+) [1,2,3] [4,5,6]`	⇝	`[5,7,9]`
`zipWith (-) [1,2,3] [4,5,6]`	⇝	`[-3,-3,-3]`
`zipWith (*) [1,2,3] [4,5,6]`	⇝	`[4,10,18]`

Table 8.3: Examples of `zip` and `zipWith`.

The Prelude function

```
unzip :: [(a, b)] -> ([a], [b])
```

takes a lists of pairs and turns it into a pair of lists.

One use for a list of pairs is a lookup table. In a lookup table, the first item of each pair serves as a *key* and the second item of each pair serves as a *value*. Such a pair is referred to as a *key-value pair*. The following list of pairs is intended to serve as a lookup table.

```
grades :: [(String, Int)]
grades = [ ("Albert Einstein", 89)
         , ("Isaac Newton"   , 95)
         , ("Alan Turing"    , 91)
         ]
```

The Prelude function

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

is designed to be given a key and a lookup table, and will return the corresponding value, if there is one.

8.7 Exercises

Exercise 8.1. Write a function

```
polarToCart :: (Double,Double) -> (Double,Double)
```

that takes as input polar coordinates (r, θ), with θ in radians, and returns as output a pair (x, y) of Cartesian coordinates.

Exercise 8.2. What are the types of `fst` and `snd`? Do these types make sense?

Exercise 8.3. What are the types of `curry` and `uncurry`?

Exercise 8.4. The Prelude function

```
head :: [a] -> a
```

is slightly problematic in that it causes a run-time error if it is passed an empty list. Write a function

```
headSafe :: [a] -> Maybe a
headSafe = undefined
```

that returns `Nothing` if passed the empty list and `Just x` otherwise, where `x` is the first element (the head) of the given list.

Exercise 8.5. We mentioned above that the type `Maybe a` is a bit like the type `[a]`, except that elements of `Maybe a` are constrained to have zero or one element. To make this analogy precise, write a function

```
maybeToList :: Maybe a -> [a]
maybeList = undefined
```

that makes a list out of a `Maybe` type. What list should `Nothing` map to? What list should `Just x` map to?

Exercise 8.6. Find out and explain what happens when `zip` is used with two lists that don't have the same length.

Exercise 8.7. Define a function

```
zip' :: ([a], [b]) -> [(a, b)]
zip' = undefined
```

that turns a pair of lists into a list of pairs. (Hint: consider using `curry` or `uncurry`.)

Exercise 8.8. The dot operator `(.)` is for function composition. If we do `unzip` followed by `zip'`, we have a function with the following type signature.

```
    zip' . unzip :: [(a, b)] -> [(a, b)]
```

Is this the identity function? (In other words, does it always return the expression it was given?) If so, how do you know? If not, give a counterexample.

If we do `zip'` followed by `unzip`, we have a function with the following type signature.

```
unzip . zip' :: ([a], [b]) -> ([a], [b])
```

Is this the identity function?

Exercise 8.9. Using the `grades` lookup table above, show how to use the `lookup` function to produce the value `Just 89`. Also show how to use the `lookup` function to produce the value `Nothing`.

Exercise 8.10. Translate the following mathematical function into Haskell.

$$x(r, \theta, \phi) = r \sin\theta \cos\phi$$

Use a triple for the input to the function `x`. Give a type signature as well as a function definition.

Chapter 9

List Comprehensions

9.1 Mapping

Haskell offers a powerful way to make new lists out of old lists. Suppose you have a list of times (in seconds)

```
ts :: [Double]
ts = [0,0.1..6]
```

and you want to have a list of positions for a rock that you threw up in the air at 30 m/s, each position corresponding to one of the times in the time list. In Exercise 2.2, you wrote a function `yRock30` to produce the position of the rock when given the time. Perhaps your function looked something like the following.

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

The code below produces the desired list of positions.

```
xs :: [Double]
xs = [yRock30 t | t <- ts]
```

The definition of `xs` is an example of a *list comprehension*. It says that for each `t` in `ts`, it will compute `yRock30 t` and form a list of the resulting values. The list `xs` of positions will be the same length as the original list `ts` of times.

Notice that we can create the same list as `xs` using `map`.

```
xs' :: [Double]
xs' = map yRock30 ts
```

We will find use for the list comprehension in forming lists of pairs (x, y) of numbers we want to plot. In Chapter 10, we will meet a plotting function `plotPath` that takes as input a list of pairs of numbers, usually `[(Double,Double)]`, and produces a plot. We can use list comprehensions to transform our data into a form suitable for plotting. If we wanted to plot position as a function of time, we could form a list of time-position pairs as follows.

```
txPairs :: [(Double,Double)]
txPairs = [(t,yRock30 t) | t <- ts]
```

Again, this can also be done with `map`.

```
txPairs' :: [(Double,Double)]
txPairs' = map (\t -> (t,yRock30 t)) ts
```

9.2 Filtering

In addition to performing the function of mapping, a list comprehension can filter data based on a Boolean expression. Let's continue our example of forming a list of time-position pairs with `yRock30`. Suppose we only want to have pairs in our list while the rock is in the air $(y > 0)$.

```
txPairsInAir :: [(Double,Double)]
txPairsInAir
    = [(t,yRock30 t) | t <- [0,0.1..20], yRock30 t > 0]
```

After we give the list from which the values of `t` are to come, we put a comma and then the Boolean expression to use for filtering. The computer will form a list as before, but now only keep values for which the Boolean expression returns `True`.

We can achieve the same effect with a combination of `map` and `filter`. We can do the filtering first,

```
txPairsInAir' :: [(Double,Double)]
txPairsInAir'
    = map (\t -> (t,yRock30 t)) $
      filter (\t -> yRock30 t > 0) [0,0.1..20]
```

or we can do the mapping first.

```
txPairsInAir'' :: [(Double,Double)]
txPairsInAir''
    = filter (\(_t,y) -> y > 0) $
      map (\t -> (t,yRock30 t)) [0,0.1..20]
```

The application operator `$` from Table 1.2 has a precedence of 0, so the expressions on each side of it are evaluated before they are combined. In this way, the application operator serves as a kind of one-symbol parentheses. The same effect could have been produced by enclosing the entire `map` line above in parentheses.

Note the use of the underscore `_` in the anonymous function above. Since the conditional expression only depends on the second item in the pair, there is no need to give a name to the first item in the pair.

The following three anonymous functions are equivalent from the perspective of the compiler.

(a) `\(t,y) -> y > 0`

(b) `\(_,y) -> y > 0`

(c) `\(_t,y) -> y > 0`

Anonymous function (a) gives the name `t` to the first item in the pair that it acts on. Since the conditional that follows doesn't use `t`, it was unnecessary to name it. Anonymous function (b) gives no name to the first item in the pair that it acts on. We are showing in our code that we don't care about that first item. Anonymous function (c) gives no name to the first item in the pair that it acts on. The compiler ignores characters that come immediately after the underscore. I used `_t` above to remind myself that the first item in the pair represents time, and also to remind myself that the time is not used in the conditional expression. Since the three expressions `t`, `_`, and `_t` produce the same effect, you are free to choose the style you like best.

9.3 Exercises

Exercise 9.1. Write the following function using a list comprehension rather than a map.

```
pick13 :: [(Double,Double,Double)] -> [(Double,Double)]
pick13 triples = map (\(x1,_,x3) -> (x1,x3)) triples
```

Exercise 9.2. List comprehensions can be used as an alternative to the `map` function. To prove this, write a function

```
  map' :: (a -> b) -> [a] -> [b]
```

That does the same thing as `map`. Use a list comprehension to write your definition.

Exercise 9.3. Suppose we throw a rock straight up in the air at 15 m/s. Use a list comprehension to make a list of (time, position, velocity) triples (type `[(Double,Double,Double)]`) for an interval of time while the rock is in the air. Your list should have enough triples so that the data would make a reasonably smooth graph if it were to be plotted.

Exercise 9.4. List comprehensions can be used as an alternative to the `filter` function. To prove this, write a function

```
  filter' :: (a -> Bool) -> [a] -> [a]
```

That does the same thing as `filter`. Use a list comprehension to write your definition.

Chapter 10

Presentation Plotting

When you make a graph for a formal report, you want to have titles, axis labels, and perhaps other features that will help the reader understand what you are trying to say.

10.1 Title and axis labels

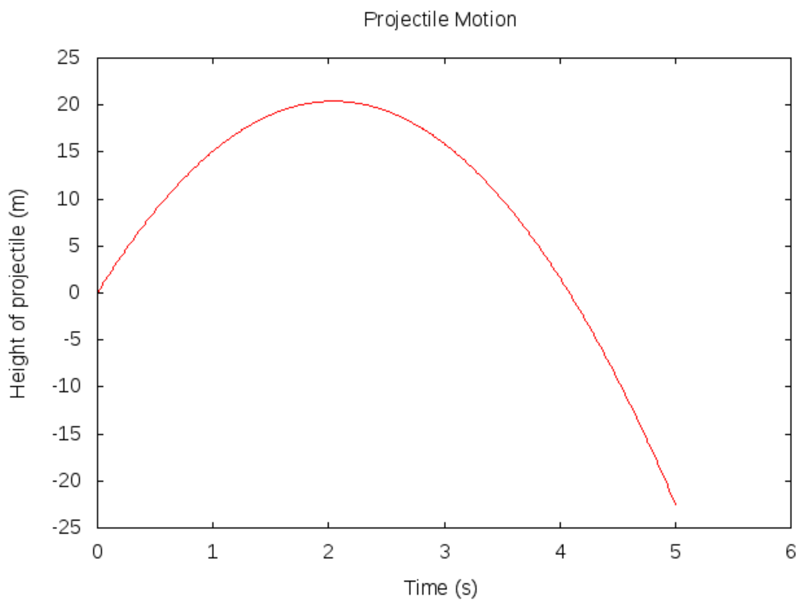
Consider the following code.

```haskell
{-# OPTIONS_GHC -Wall #-}

import Graphics.Gnuplot.Simple

type R = Double

tRange :: [R]
tRange = [0,0.01..5]

yPos :: R  -- y0
     -> R  -- vy0
     -> R  -- ay
     -> R  -- t
     -> R  -- y
yPos y0 vy0 ay t = y0 + vy0 * t + ay * t**2 / 2
```

```
plot1 :: IO ()
plot1 = plotFunc [Title "Projectile Motion"
                 ,XLabel "Time (s)"
                 ,YLabel "Height of projectile (m)"
                 ,PNG "projectile.png"
                 ,Key Nothing
                 ] tRange (yPos 0 20 (-9.8))
```

The very first line turns on warnings, which I like to do to catch any poor programming I might not have intended. The second line imports the `Graphics.Gnuplot.Simple` module, which we use to make plots. The third line sets up `R` as a *type synonym* for `Double`, so that I can think of `Double`s as real numbers, and call them by the short name `R`. We then define a list `tRange` of time values that we will use in our plot. We define a function `yPos` for the height of a projectile. Finally, we define `plot1` to make a plot. Notice the type `IO ()` (pronounced "eye oh unit") of `plot1`. `IO` is a type constructor (like `Maybe`) that has the special job of signaling that `plot1` returns a result of type `()` (pronounced "unit") and also performs some effect, in this case producing a file containing a plot. The type `()` only contains one term `()` (also called "unit"), so it can't convey any actual information. Anything with type `IO ()` is something that is done only for its effect.

If you load this code into GHCi and type `plot1` at the prompt, it will produce a file `projectile.png` on your hard drive that you can include in a document. Here is what it looks like.



Recall that `plotFunc` has type `[Attribute] -> [a] -> (a -> a) -> IO ()` where `a` is a type in some specialized type classes. The `Attribute` type is defined in the `Graphics.Gnuplot.Simple` module. If you type `:i Attribute` at the GHCi prompt, (`:i` is short for `:info`) you'll see some options for what you can do with these `Attribute`s. Function `plotFunc` takes a list of `Attribute`s, and we are putting items in this list to produce the title and axis labels.

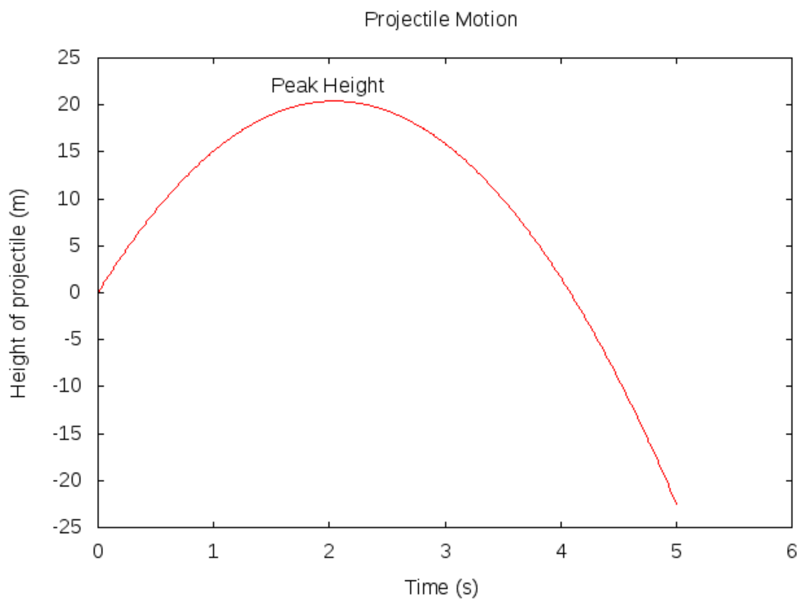
10.2 Other labels

You may want to put other labels on a plot. Here is how you can do this.

```
plot1Custom :: IO ()
plot1Custom
    = plotFunc [Title "Projectile Motion"
               ,XLabel "Time (s)"
               ,YLabel "Height of projectile (m)"
               ,PNG "CustomLabel.png"
               ,Key Nothing
```

```
                ,Custom "label" ["\"Peak Height\" at 1.5,22"]
                ] tRange (yPos 0 20 (-9.8))
```

Note the `Custom` attribute that we added. The backslash in front of the quotes is because we need to pass quotes inside of quotes. The coordinates 1.5,22 are the horizontal and vertical coordinates on the graph where we want the label to appear. Here is what it looks like.



10.3 Plotting data

There will be times when we want to plot points of (x, y) pairs rather than functions. We can use the `plotPath` function for this (also defined in `Graphics.Gnuplot.Simple`).

10.4 Multiple curves on one set of axes

You can plot multiple curves on a single set of axes. This is particularly useful if you want to compare two functions that have the same independent and dependent variables. Consider the following plot.

```
plot2 = plotFuncs [] [0,0.1..10] [cos,sin]
```

Try to load this into GHCi.

Activity 10.1. Add one or more type signatures to this program so that it correctly loads and makes a plot.

Notice that the `plotFuncs` function takes a list of functions as one of its arguments. We found a use for a list of functions!

The range of x-values does not have to be the same for the two plots. Consider the following example, which introduces the new function `plotPaths`. Can you see how `plotPaths` differs from `plotPath`?

```
xs1, xs2 :: [R]
xs1 = [0,0.1..10]
xs2 = [-5,-4.9..5]

xys1, xys2 :: [(R,R)]
xys1 = [(x,cos x) | x <- xs1]
xys2 = [(x,sin x) | x <- xs2]

plot2 :: IO ()
plot2 = plotPaths [] [xys1,xys2]
```

You can plot three things at the same time if you like.

```
xRange :: [R]
xRange = [0,0.02..10]

f3 :: R -> R
f3 x = exp (-x)

plot3 :: IO ()
plot3 = plotFuncs [] xRange [cos,sin,f3]
```

10.5 Controlling the plot ranges

By default, Gnuplot (the program that is making the graphs behind the scenes) will make plots based on the x-ranges that you provide, and the corresponding y-ranges that are calculated. Sometimes, you may want more control over the x-range or the y-range.

Revisiting the previous example of three plots, try the following.

```
plot3' :: IO ()
plot3' = plotFuncs [ XRange (-2,8)
                   , YRange (-0.2,1)
                   ] xRange [cos,sin,f3]
```

Notice the funny stylistic way in which I made the list `[XRange (-2,8), YRange (-0.2,1)]`. People who code in Haskell sometimes do this (putting the comma first on the second line of the list), but you don't have to. You could put this all on one line, or put the comma at the end of the line. It's a matter of style.

Activity 10.2. Write a function

```
approxsin :: R -> R
approxsin = undefined
```

that approximates the sine function by the first four terms in its Taylor expansion,

$$x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}.$$

(Depending on how you do this, you may or may not run into the issue that you cannot divide a `Double` by an `Int` or an `Integer` in Haskell. You can only divide a numeric type by the same numeric type. If you run into this problem, you can use the function `fromIntegral` to convert an `Int` or an `Integer` to some other type, like `Double`.)

Test your function by trying the following command in GHCi.

- `plotFuncs [] [-4,-3.99..4] [sin,approxsin]`

Make a nice version of this plot (with a title, axis labels, labels to indicate which curve is which, etc.).

10.6 Exercises

Exercise 10.1. Make a plot of $y = x^2$ from $x = -3$ to $x = 3$ with a title and axis labels.

Exercise 10.2. Take a look at the type signature for `plotPath`, and figure out how to plot the list of points `txPairs` below.

```
ts :: [Double]
ts = [0,0.1..6]

txPairs :: [(Double,Double)]
txPairs = [(t,30 * t - 4.9 * t**2) | t <- ts]
```

Make a plot with a title and axis labels (with units).

Chapter 11

Animation

The Haskell Prelude itself does not have any support for animation, but there are good libraries available. For two-dimensional pictures and animations, we will use the *gloss* library. For three-dimensional pictures and animations, we will use a library named *not-gloss*.

11.1 2D Animation

The *gloss* library provides four main functions: `display`, `animate`, `simulate`, and `play`. The first is for still pictures, the second and third are for pictures which change with time, and the fourth is for pictures which change with time and user input. We are interested primarily in the first three functions.

11.1.1 Displaying a picture

The function `display` produces a static picture. Let's ask GHCi for the type of `display`. Since `display` is not part of the Prelude, we must load the module.

GHCi
```
:m Graphics.Gloss
```

GHCi
```
:t display
    ⇝  display :: Display -> Color -> Picture -> IO ()
```

The types `Display`, `Color`, and `Picture` are defined by the *gloss* library, and `IO ()` (pronounced "eye oh unit") is the type given to a Haskell *action*, which means the computer is going to *do* something (display a picture) rather than return a value. `IO` is a type constructor, like `Maybe`, but it is a special type constructor that is designed to signal an *effect*, which is a computation that is not purely functional. An effect changes the world in some way. It is more than a calculation. A file on the harddrive changing is an effect. A picture being shown on the screen is an effect.

The types `Display` and `Color` are for display mode and background color, respectively. The most interesting type is `Picture`, which represents the type of things that can be displayed. The *gloss* documentation on `Picture` describes the pictures that can be made (lines, circles, polygons, etc.).

GHCi is not so good at showing the pictures that *gloss* creates, so it is better to make a stand-alone program. A minimal stand-alone program (from the *gloss* documentation) is as follows.

```
{-# OPTIONS_GHC -Wall #-}

import Graphics.Gloss

main :: IO ()
main = display (InWindow "Nice Window" (200, 200) (10, 10))
       white (Circle 80)
```

A stand-alone program must have a `main` function, which is the function that will be evaluated when the program is run. Suppose we put this code in a file named `MinimalGloss.hs`. To compile the stand-alone program, issue the command

```
ghc --make MinimalGloss.hs
```

from the command line (not from inside GHCi). This should produce an executable program named `MinimalGloss` which can be run.

Activity 11.1. Consult the *gloss* documentation on the `Picture` type and make an interesting picture using the `display` function. Combine lines, circles, text, colors, and whatever you like. Be creative.

11.1.2 Making an animation

Given a picture as a function of time, the function `animate` produces an animation. The type of `animate` is the following.

```
animate :: Display -> Color -> (Float -> Picture) -> IO ()
```

The difference in type compared to `display` is that the `Picture` in `display` has been replaced by a `Float -> Picture` in `animate`. The `animate` function uses a `Float` to describe time, so an expression of type `Float -> Picture` is a function from time to pictures, or a picture as a function of time.

Activity 11.2. Use `animate` to make a simple animation. Be creative.

11.1.3 Making a simulation

The *gloss* function `simulate` allows the user to make an animation in which an explicit function describing a picture as a function of time is not available. The type of `simulate` is more complicated, and will be easier to describe after we do a little physics. The `simulate` function is probably the most useful for physics of all the *gloss* functions.

11.2 3D Animation

The *not-gloss* library provides four main functions whose names are identical to those in *gloss*: `display`, `animate`, `simulate`, and `play`. As in *gloss*, the first is for still pictures, the second and third are for pictures which change with time, and the fourth is for pictures which change with time and user input. We are interested primarily in the first three functions. The types of these functions are different from the types of the corresponding *gloss* functions. To use the *not-gloss* library, you must include

```
import Vis
```

at the top of you Haskell source file.

Part II

Newtonian Mechanics

Chapter 12

Newton's Second Law

12.1 Newton's First Law

Have you ever seen an air track? An air track is a fun toy (or piece of experimental equipment if you're feeling more serious) that consists of a long horizontal rail (maybe 2 or 3 meters long) with little holes that allow air to shoot upward out of the rail. A small car (maybe 5 cm wide and 10 cm long) rides atop this air track. The purpose of the air is to eliminate most of the friction that would exist between the car (which has no wheels) and the rail as the car slides along the rail, so that the car can slide quite freely along the air track. The cross section of the rail is shaped so that the car can only slide back and forth along the length of the rail; the car cannot slide sideways and it cannot move upward or downward in height.

If you give the car a little push on the air track and then let it go, you can see that it travels at a constant speed until it hits the end of the track (nice cars have springy bumpers that allow the car to make a gentle bounce at the end of the track and slide in the opposite direction). 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*. In fact, this idea is important enough to be enshrined in a principle of physics called *Newton's First Law*. Here are two versions of Newton's First Law.

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 speed in the same direction.*

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*, then the speed will stay constant now. Any time there are no forces present, the speed will stay constant.

Question 12.1. Suppose we have an air track, and a car with bumpers on both ends, so that the car will bounce when it gets to either end of the air track. We give the car a little push and then let it go. On a real air track in the lab, the car may bounce at one end, then return to the other end and bounce again, but there is a little bit of friction that the air can't completely eliminate, and the car will eventually slow down and stop. On an ideal (perfect) air track that we construct in our minds, we suppose that we can completely eliminate friction, and also that the bounces at the ends allow the car to keep the same speed, only in the reverse direction. If we give our car a little push on this ideal air track, how many times will it bounce at each end of the track?

12.2 Newton's Second Law

Newton's First Law tells us what happens when no forces are present. Conversely, any time forces act on an object, the speed of the object may very well change. Large forces can give rise to rapid changes in speed, while small forces are likely to cause speed to change more slowly.

We can make the relationship between applied forces and change in speed more precise. In order to do so, we need to develop some careful vocabulary.

First, we need to make a distinction between *velocity* and speed. Speed is simply a number with units, such as 30 m/s. (The units m/s mean meters per second.) Speed is never negative. Velocity includes more information than speed. Velocity incorporates the direction of motion as well as the speed. Suppose our air track is situated in our lab with one end toward the North and the other end toward the South. We can choose a coordinate system in which one of these directions is regarded as the positive direction and the other is regarded as the negative direction. There are two coordinate systems to choose from. We could choose North as the positive direction or we could choose South as the positive direction. Let us choose North as the positive direction and South as the negative direction. A velocity of -20 m/s means a speed of 20 m/s in the South direction. In this chapter and the next, we consider the simplified case of motion in one dimension, such as along an air track. For motion in one dimension, positive and negative numbers are sufficient to describe velocity. For motion in two or three dimensions, which we begin to consider in Chapter 14, we will need vectors to describe velocity. For one-dimensional motion, we will use the symbol v for velocity.

Acceleration describes how velocity changes with time. Acceleration is the key concept we need to be able to say how an object responds to the application of forces. Let us consider a small interval of time over which there may be forces acting on our car on the air track. Let t_0 be the time at the beginning of the time interval and v_0 be the velocity of the car at time t_0. Let t_1 be the time at the end of the time interval and v_1 be the velocity of the car at time t_1. (If there were no forces acting during the time interval from t_0 to t_1, then Newton's First Law would tell us that the velocity stays the same, that is $v_0 = v_1$.) The length of our time interval is

$$\Delta t = t_1 - t_0.$$

In the general case where forces are present, the velocity can change. The change in velocity over the course of the time interval is

$$\Delta v = v_1 - v_0.$$

The *average acceleration* over the time interval Δt is defined to be

$$\overline{a} = \frac{\Delta v}{\Delta t} = \frac{v_1 - v_0}{t_1 - t_0}.$$

We use the symbol $\overline{a}$ for average acceleration.

We would very much like to be able to speak of the acceleration at a particular point in time rather than the (average) acceleration over a time interval. The purpose of calculus is to give us a language and a set of ideas that allow us to define things like *instantaneous acceleration*, which is acceleration at a point in time. Roughly speaking, we want to allow the time interval Δt to shrink toward zero as we "take the limit" by observing what happens to the ratio $\Delta v/\Delta t$ as $\Delta t \to 0$. In the notation of calculus, we write

$$a = \frac{dv}{dt} \tag{12.1}$$

in which a is the instantaneous acceleration, and dv/dt is called the derivative of velocity with respect to time. In words, we say any of the following.

- Acceleration is the derivative of velocity with respect to time.
- Acceleration is the rate of change of velocity with respect to time.
- Acceleration is the slope of the velocity-time graph.

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 of the forces acting on the object. (In one dimension, some forces may be negative and some may be positive.)

Newton's Second Law in one dimension

$$a = \frac{F_{\text{net}}}{m} \tag{12.2}$$

```
{-# OPTIONS_GHC -Wall #-}

module Newton2 where

import Graphics.Gnuplot.Simple

type R = Double
```

Example 12.1. Suppose we have a car with mass 0.1 kg on an air track. The car is initially moving to the North 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 (N is an abbreviation for

Newton, the unit of force) to the North. At the same time, our friend applies to the same car a constant force of 0.08 N to the South. (By "constant", we mean that the force continues to exist with the same value for some period of time.) What will the subsequent motion of the car look like? In particular, how will the velocity of the car change in time?

We can use Newton's Second Law to calculate the acceleration of the car over the period of time during which the forces act. Taking North to be the positive direction, we have $F_1 = 0.04$ N and $F_2 = -0.08$ N. Therefore,

$$F_{\text{net}} = F_1 + F_2 = (0.04\text{ N}) + (-0.08\text{ N}) = -0.04\text{ N}.$$

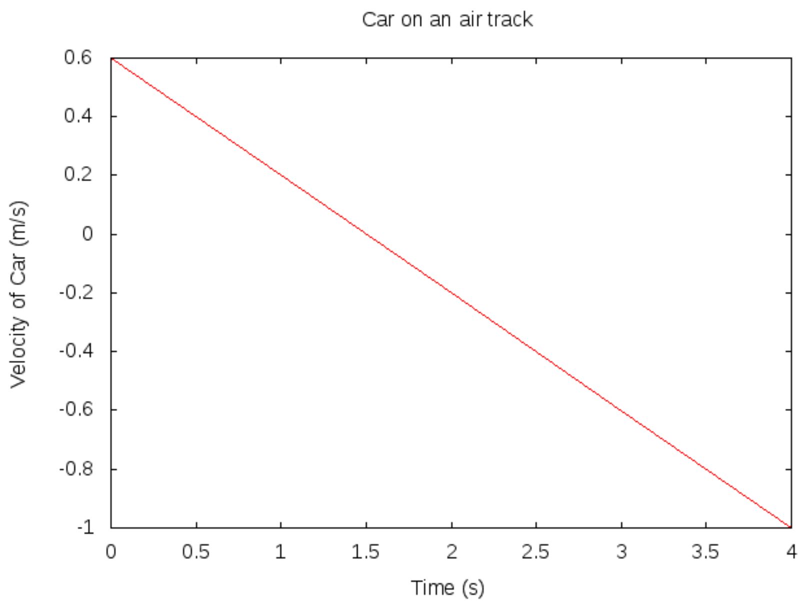
$$a = \frac{F_{\text{net}}}{m} = \frac{-0.04\text{ N}}{0.1\text{ kg}} = -0.4\text{ m/s}^2$$

Because the forces are constant, the acceleration is also constant. Since acceleration is the rate of change of velocity, an acceleration of -0.4 m/s^2 means that the velocity changes by 0.4 m/s to the South every second. We can write an equation for the subsequent velocity.

$$v(t) = 0.6\text{ m/s} - (0.4\text{ m/s}^2)t \tag{12.3}$$

We can make a graph of the velocity of the car as a function of time.

```
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 :: R] (\t -> 0.6 - 0.4 * t)
```



Note that a negative acceleration (which we have over the entire time interval from $t = 0$ to $t = 4$ s) does not necessarily mean that the car is slowing down. Rather, a negative acceleration means an acceleration to the South. The car slows down during the first 1.5 s as it is moving North, but then begins to speed up as it moves South. 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.

Example 12.1 showed a situation in which multiple forces acted on an object, but all of the forces were constant in time. The forces did not change over time, and so the acceleration was also constant. The acceleration did not change over time. In a typical introductory physics course, you see a lot of problems with constant acceleration, because these are the problems that are readily solved without a computer. What kind of situation would fail to have constant acceleration?

Example 12.2. Let's consider a bicycle rider riding North on a flat level road. We will 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. Second, there is the southward force of air resistance that impedes the northward progress of the rider, especially when she is traveling fast. The force of air resistance is a good example of a non-constant force. The air resistance depends on the speed of the bike through the air. The force of air resistance is larger when the bicycle speed is larger. There is no simple, universal law for the force of air resistance. It is a complicated fluid dynamics problem in general (gases and liquids are fluids, so air is a fluid). Let's suppose in our problem a simple relationship between the bike speed and the force of air resistance. The simplest relationship would be if the force of air resistance were proportional to the bike speed. Twice the speed would result in twice the force. We could write

$$F_{\text{air}} = Av$$

where A is a constant of proportionality that depends on the size and shape of the bike, the temperature and humidity of the air, and maybe other factors. We're not going to get into any of these factors, we're just going to say that A is some constant. The constant A is related to what people sometimes call a *drag coefficient*. Let us call the force produced by the rider 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 constant. Let's take the mass of the bike plus rider to be $M = 70$ kg. Newton's second law tells us how to find acceleration.

$$a = \frac{F_{\text{rider}} - F_{\text{air}}}{M}$$

We choose North to be the positive direction, so air resistance needs to come in with a minus sign. Substituting our expression for air resistance, we have

$$a = \frac{F_{\text{rider}} - Av}{M}.$$

Now *this* is an interesting situation. Acceleration tells velocity how to change. If we know the acceleration, we know the rate at which velocity changes. But in the present case, acceleration depends on velocity. We seem to have a sort of chicken-and-egg problem. We need acceleration to find velocity and we need velocity to find acceleration. The whole subject of differential equations was invented to help with situations like this. We are not going to trot off into the world of differential equations. It is true that differential equations are secretly behind the methods that we will develop, but we will gently tap their power and not shout about them too much right now.

12.3 State update

The key idea that will get us some traction out of the chicken-and-egg problem is to think of the velocity of the bike as part of the *state* of the bicycle at any particular moment of time. For this situation, we are going to take the state of the bicycle to consist of two items: time on the clock and velocity of the bike.

We will further imagine that at the beginning of the situation (perhaps when the rider starts riding) we know the initial state of the bike. Then what Newton's second law gives us is a *state update rule*. Newton's second law will tell us how to advance from

$$(t, v) \to (t', v')$$

or, better yet, from

$$(t, v) \to (t + \Delta t, v + \Delta v).$$

The Δt is a small time step that we will choose. We should pick it to be "small" in the context of the situation. For a rider on a bicycle, $\Delta t = 1$ s is probably small enough. Interesting changes do not occur over a one second interval when riding a bike. (If there was a crash, that would be an exception, and we would want to use a smaller time step to analyze that, but notice that a crash would also require at least one more force.)

Accleration tells velocity how to change, and Newton's second law says that acceleration is net force over mass, so our state update equations are

$$\begin{aligned} t' &= t + \Delta t \\ v' &= v + \frac{F_{\text{net}}}{m}\Delta t. \end{aligned} \tag{12.4}$$

Applying state update equations like this is sometimes called 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.

Example 12.3. For the rider on the bicycle, use the state update equations with $F_{\text{rider}} = 100$ N, $A = 10$ kg/s, $M = 70$ kg, and a time step $\Delta t = 1$ s to complete the following table.

t (s)	v (m/s)
0	0.0000
1	
2	
3	

Solution: The state update equations are

$$t' = t + \Delta t$$
$$v' = v + \frac{F_{\text{rider}} - Av}{M}\Delta t.$$

We put in the state variables in the first row of the table to obtain the state variables in the second row of the table.

$$t' = 0 + 1 = 1$$
$$v' = 0.0000 + \frac{100 - 10 * 0.0000}{70} * 1 = 1.4286$$

We update the table.

t (s)	v (m/s)
0	0.0000
1	1.4286
2	
3	

Now we update the state variables in the second row of the table to obtain the state variables in the third row of the table.

$$t' = 1 + 1 = 2$$
$$v' = 1.4286 + \frac{100 - 10 * 1.4286}{70} * 1 = 2.6531$$

We update the table.

t (s)	v (m/s)
0	0.0000
1	1.4286
2	2.6531
3	

Now we update the state variables in the third row of the table to obtain the state variables in the fourth row of the table.

$$t' = 2 + 1 = 3$$
$$v' = 2.6531 + \frac{100 - 10 * 2.6531}{70} * 1 = 3.7027$$

We can now complete the table.

t (s)	v (m/s)
0	0.0000
1	1.4286
2	2.6531
3	3.7027

Let's write some code for the bike situation.

```
bikeStateUpdate :: R       -- time step
                -> (R,R)   -- starting state
                -> (R,R)   -- ending state
bikeStateUpdate dt (t,v) = (t + dt, v + dv)
    where
      dv = acceleration * dt
      acceleration = (forceRider - forceAir) / mass
      mass = 70
      forceRider = 100
      forceAir = drag * v
      drag = 10
```

We now want to link up a whole chain of these state updates to get a sense of what is happening over time.

```
bikeSolution :: R         -- time step
             -> (R,R)     -- initial state
             -> [(R,R)]   -- infinite list of future states
bikeSolution dt = iterate (bikeStateUpdate dt)
```

There are other ways to write this definition. Here is a list of equivalent definitions.

(a) `bikeSolution dt = iterate (bikeStateUpdate dt)`

(b) `bikeSolution dt = iterate $ bikeStateUpdate dt`

(c) `bikeSolution dt = (iterate . bikeStateUpdate) dt`

(d) `bikeSolution = \dt -> iterate (bikeStateUpdate dt)`

(e) `bikeSolution = \dt -> iterate $ bikeStateUpdate dt`

(f) `bikeSolution = \dt -> (iterate . bikeStateUpdate) dt`

(g) `bikeSolution = iterate . bikeStateUpdate`

Definition (c) has the property that `bikeSolution` is being defined by acting directly on `dt` with the function `(iterate . bikeStateUpdate)`. When this happens, the `dt` can be removed from both sides if you wish. Doing this is called *point-free style*. It is just a stylist choice. Some people like it and find it insightful. Others find it opaque. Feel free to use it if you like it and avoid it if you don't. You can see that items (d)–(f) are slight modifications of items (a)–(c), respectively, in which an anonymous function is used on the right side instead of mentioning the argument on the left side. Item (g), the point-free style, can be thought of as replacing the anonymous function `\dt -> (iterate . bikeStateUpdate) dt` with the function `iterate . bikeStateUpdate`. This makes sense. It's like saying the anonymous function `\x -> sqrt x` is the same as the function `sqrt`. It is.

Here is the function in point-free style.

```
bikeSolution' :: R          -- time step
              -> (R,R)      -- initial state
              -> [(R,R)]    -- infinite list of future states
bikeSolution' = iterate . bikeStateUpdate
```

Let's use these functions to check the work we did by hand above.

```
GHCi  :l Newton2.lhs
```

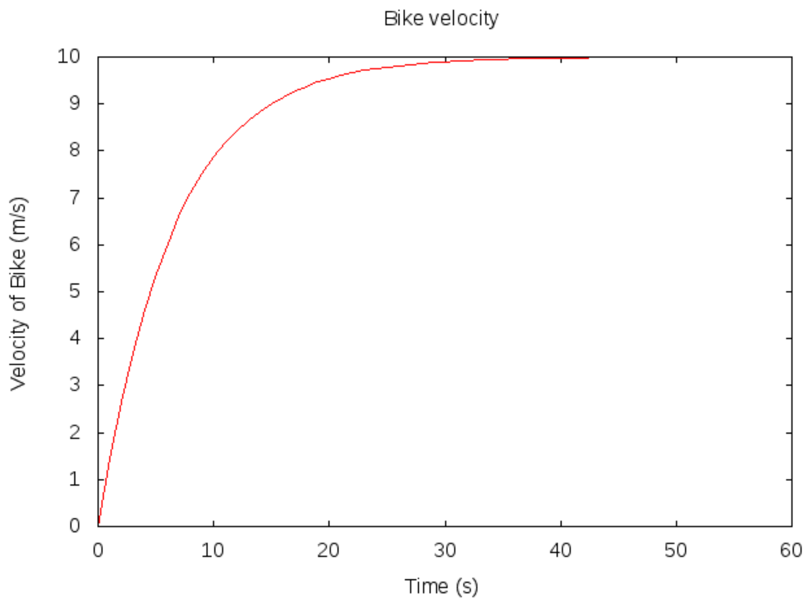
```
GHCi  bikeSolution 1 (0,0) !! 3
          ⇝  (3.0,3.702623906705539)
```

This time and velocity match the last row of our table very nicely.

We can make a graph of velocity as a function of time.

```
bikeGraph :: IO ()
bikeGraph = plotPath [Title "Bike velocity"
                     ,XLabel "Time (s)"
                     ,YLabel "Velocity of Bike (m/s)"
```

```
                   ,PNG "BikeVelocity.png"
                   ,Key Nothing
                   ] (takeWhile (\(t,_) -> t <= 60)
                      (bikeSolution 1.0 (0,0)))
```



A phenomenon occurs here that does not occur in constant acceleration situations, and that is the establishment of a terminal velocity. After 30 seconds or so, the forward force of the road (or 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.

Activity 12.1. (Euler method by hand 1) To deepen our understanding of the Euler method, we'll do a calculation by hand (using only a calculator, and not the computer).

Consider the differential equation

$$\frac{dv}{dt} = -3v + 4\cos 2t$$

for the function v, along with the initial condition

$$v(0) = 2.$$

Use the Euler method with a step size of $\Delta t = 0.1$ to approximate the value of $v(0.3)$.

Keep at least four figures after the decimal point in your calculations. Show your calculations in a small table.

Let (t, v) be a pair of values representing the independent (t) and dependent (v) variables at a moment in time. Let us call this pair a *state tuple*, because it encapsulates the state of the system at a moment in time. A helpful way of thinking about a single step of the Euler method is that it takes a state tuple (t, v) as input and gives as output a new state tuple (t', v') representing the variables at a later point in time (the step size Δt later in time).

$$(t, v) \to (t', v')$$

The Euler method prescribes that the new values are related to the old values by

$$\begin{aligned} t' &= t + \Delta t \\ v' &= v + a(t, v)\Delta t \end{aligned}$$

where $a(t, v)$ is the acceleration as a function of t and v.

Activity 12.2. (Euler method by computer 1) Write a Haskell function

```haskell
eulerStepExample :: (R,R) -> (R,R)
eulerStepExample = undefined
```

that takes a pair (t, v) and returns a pair (t', v') for a single step of the Euler method for the differential equation

$$\frac{dv}{dt} = -3v + 4\cos 2t$$

with a step size of $\Delta t = 0.1$.

Show how to use the function `eulerStep1` to calculate the value $v(0.3)$ that you calculated by hand before.

```
eulerStep1 :: R
           -> ((R,R) -> R)
           -> (R,R) -> (R,R)
eulerStep1 dt a (t,v) = (t',v')
  where
    t' = t + dt
    v' = v + a(t,v) * dt
```

Activity 12.3. Consider the differential equation

$$\frac{dv}{dt} = -v$$

subject to the initial condition $v(0) = 8$. Solve this initial value problem over the time interval $0 \leq t \leq 10$ using the Euler method. (You will probably want to use the Prelude function `iterate`.) Plot v as a function of t to see what it looks like. Compare your results to the exact solution of the differential equation (You can solve this differential equation exactly by guessing or some other method.) Try out different time steps to see what happens when the time step gets too big.

Find a time step that is small enough so that the Euler solution and the exact solution overlap precisely on a plot. Find another time step that is big enough so 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, etc.) 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". Put your name on the plot (using the label commands we learned).

Activity 12.4. Consider the differential equation

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

subject to the initial condition $v(0) = 0$. You cannot solve this differential equation by hand. (Why not?) 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 on your plot (using a label command) the value $v(3)$ to five significant figures. Put your name on the plot (using a label command).

Chapter 13

Mechanics in One Dimension

Mechanics is easiest in one spatial dimension, because quantities like position, velocity, acceleration, force, and momentum can be represented by numbers rather than by vectors.

13.1 State update

In Chapter 12, we introduced the idea that Newton's second law is a state update rule. We used a time-velocity pair (t, v) to characterize the state of an object such as a car or a bicycle that can move in one dimension. We used Newton's second law to answer the question of how to update the state

$$(t, v) \to (t', v')?$$

The new time t' is the old time t plus a change in time. The new velocity v' is the old velocity v plus a change in velocity.

$$\begin{aligned} t' &= t + \Delta t \\ v' &= v + \Delta v \end{aligned}$$

If Δt is small compared with the important time scales in the problem, then $\Delta v/\Delta t$ approximates the instantaneous acceleration the object experiences, which by Newton's second law is the net force on the object divided by its mass.

$$\frac{\Delta v}{\Delta t} = \frac{F_{\text{net}}}{m}$$

Newton's second law provides a way to approximate Δv for an object if we know all the forces that act on the object.

$$\Delta v = \frac{F_{\text{net}}}{m} \Delta t$$

Here, then, are our state update equations.

$$\begin{aligned} t' &= t + \Delta t \\ v' &= v + \frac{F_{\text{net}}}{m} \Delta t \end{aligned} \qquad (13.1)$$

We assume that we know the current state (t, v) of the object of interest. We assume that we know the mass m of the object, and that it is constant. We also assume that from (t, v) we can get numbers for all of the forces that act on the object. If these assumptions all hold, then the state update equations will give us a new state (t', v') consisting of a new time t' (just a little later than the old time) and a new velocity v'. (Remember velocity remains constant under conditions of no net force, but changes when a net force is present.)

One of the assumptions we just mentioned is too hopeful and usually not true. It is the assumption that if we know t and v, we can get numbers for all of the forces that act on an object. That assumption held for all of the examples in Chapter 12, but it leaves out the possibility that a force may depend on the *position* (location) of an object. If an object is attached to a spring, for example, the force that the spring applies to the object depends on how far the spring is extended or compressed. The spring force depends on where the object is. Another force that depends on location is Newton's universal gravity, which is inversely proportional to the square of the distance from the object to another object. Again we need to know where the object is to compute the force that acts on it.

The fact that we cannot compute all of the forces that we are interested in from a state consisting of a time-velocity pair (t, v) means that the time-velocity pair is in general too impoverished to serve as the state of an object. Knowing that we are interested in forces that depend on the position of our object, we expand our state to be a time-position-velocity triple (t, x, v). This triple will serve as the state of an object for this chapter.

With a hopeful new state in hand, we return to the question of state update. How can we advance the state

$$(t, x, v) \to (t', x', v')?$$

The good news is that we can keep our state update equations (13.1), with the new understanding that the forces may depend on any of t, x, and/or v. We need an update equation for x'. The definition of velocity helps us here. If Δt is small compared with the time scales in the problem, then $\Delta x/\Delta t$ is a good approximation to the instantaneous velocity of our object. In other words, $\Delta x = v\Delta t$. Here are our new state update equations.

$$\begin{aligned} t' &= t + \Delta t \\ x' &= x + v\Delta t \\ v' &= v + \frac{F_{\text{net}}}{m}\Delta t \end{aligned}$$

Example 13.1. (State update by hand.)

Consider a damped, driven harmonic oscillator. This is a mass m on the end of a spring that can oscillate horizontally on the surface of a table. There are three forces that act on the mass. The spring force is produced by the spring and acts to restore the mass to an equilibrium position. The spring force is given by Hooke's law

$$F_{\text{spring}} = -kx$$

which claims that the force produced by the spring is proportional to the displacement of the mass from its equilibrium position. The constant k is the *spring constant*. A spring with a large spring constant is a stiff spring that takes lots of force to extend or compress. The negative sign makes the spring force a restoring force. The equilibrium position is $x = 0$. If x is positive, then F_{spring} is negative, and the force acts toward the equilibrium position. If x is negative, then F_{spring} is positive, and again the force acts toward the equilibrium position. A second force is a damping force

$$F_{\text{damp}} = -bv$$

which is a frictional force proportional to velocity. It could be caused by air resistance. The damping constant b gives a sense of the size of this frictional force. The third force is the driving force

$$F_{\text{drive}} = F_0 \cos \omega_0 t$$

which is an oscillating force, applied with amplitude F_0 and angular frequency ω_0.

By including all three of these forces, we have a net force that depends on all three of our state variables (t, x, v).

Use the state update equations with $F_0 = 40$ N, $\omega_0 = 6$ rad/s, $k = 500$ N/m, $b = 20$ kg/s, $m = 5$ kg, and a time step $\Delta t = 0.1$ s to complete the following table.

t (s)	x (m)	v (m/s)
0.0	0.1000	0.0000
0.1		
0.2		
0.3		

Solution: The state update equations are

$$t' = t + \Delta t$$
$$x' = x + v\Delta t$$
$$v' = v + \frac{F_0 \cos \omega_0 t - kx - bv}{m}\Delta t$$

We put in the state variables in the first row of the table to obtain the state variables in the second row of the table.

$$t' = 0.0 + 0.1 = 0.1$$
$$x' = 0.1000 + 0.0000 * 0.1 = 0.1000$$
$$v' = 0.0000 + \frac{40\cos(6 * 0.0) - 500 * 0.1000 - 20 * 0.0000}{5} * 0.1 = -0.2000$$

We update the table.

t (s)	x (m)	v (m/s)
0.0	0.1000	0.0000
0.1	0.1000	-0.2000
0.2		
0.3		

Now we put in the state variables in the second row of the table to obtain the state variables in the third row of the table.

$$t' = 0.1 + 0.1 = 0.2$$
$$x' = 0.1000 - 0.2000 * 0.1 = 0.0800$$
$$v' = -0.2000 + \frac{40\cos(6 * 0.1) - 500 * 0.1000 + 20 * 0.2000}{5} * 0.1 = -0.4597$$

We update the table.

t (s)	x (m)	v (m/s)
0.0	0.1000	0.0000
0.1	0.1000	-0.2000
0.2	0.0800	-0.4597
0.3		

Now we put in the state variables in the third row of the table to obtain the state variables in the fourth row of the table.

$$t' = 0.2 + 0.1 = 0.3$$
$$x' = 0.0800 - 0.4597 * 0.1 = 0.0340$$
$$v' = -0.4597 + \frac{40\cos(6 * 0.2) - 500 * 0.0800 + 20 * 0.4597}{5} * 0.1 = -0.7859$$

We can now complete the table.

t (s)	x (m)	v (m/s)
0.0	0.1000	0.0000
0.1	0.1000	-0.2000
0.2	0.0800	-0.4597
0.3	0.0340	-0.7859

You can see that we are finding out how the position and the velocity of the object are changing in time. By letting the computer do this work, we can get accurate results over long periods of time.

Let's write some code so that the computer can automate the state update for us. I always like to start by turning on warnings.

```
{-# OPTIONS_GHC -Wall #-}
```

Let's make the code in this chapter into a module and give it the name `Mechanics1D`.

```
module Mechanics1D where
```

We will want to make a plot later, so let's import that module

```haskell
import Graphics.Gnuplot.Simple
```

Let's use the type synonym `R` instead of `Double`. It's shorter and it reminds us of the real numbers.

```haskell
type R = Double
```

Now there are some other type synonyms I'd like to introduce.

```haskell
type Time           = R
type TimeStep       = R
type Position1D     = R
type Velocity1D     = R
type Acceleration1D = R
```

Here we are saying that time, a time step, position, velocity, and acceleration in one dimension are all represented by a real number. The next type synonym defines our state as a time-position-velocity triple.

```haskell
type StateTuple1D = (Time,Position1D,Velocity1D)
```

Our final type synonym is more complicated. It defines something called an *acceleration function*, which is the function of t, x, and v that gives the acceleration through Newton's second law

$$a(t,x,v) = \frac{F_{\text{net}}(t,x,v)}{m}$$

where we recall that the net force can depend on t, x, and v. Since acceleration depends on the state tuple, and produces an acceleration, it is a function with the following type.

```haskell
type AccelerationFunction1D = StateTuple1D -> Acceleration1D
```

The next function is probably the most important function in this chapter. It is the state update rule for one-dimensional mechanics. This state update rule can work with *any* forces that depend only on t, x, and v (which is almost all forces).

```
eulerStep1D :: TimeStep
            -> AccelerationFunction1D
            -> StateTuple1D -> StateTuple1D
eulerStep1D dt a (t,x,v) = (t',x',v')
    where
      t' = t + dt
      x' = x + v * dt
      v' = v + a(t,x,v) * dt
```

Finally, to build a "table" of values like we did by hand in the damped, driven harmonic oscillator example, we have a function to iterate the state update.

```
eulerSolution1D :: TimeStep
                -> AccelerationFunction1D
                -> StateTuple1D -> [StateTuple1D]
eulerSolution1D dt a (t,x,v)
    = iterate (eulerStep1D dt a) (t,x,v)
```

This completes our general technology for dealing with one-dimensional mechanics. Let's now apply this technology to the damped, driven harmonic oscillator that we looked at by hand.

Once we know all of the forces that act on our object, we can write an acceleration function.

```
accelFuncDampedDrivenHarm :: AccelerationFunction1D
accelFuncDampedDrivenHarm (t,x,v)
    = (fDrive + fSpring + fDamp) / m
      where
        fDrive  = f0 * cos (omega0 * t)
        fSpring = -k * x
        fDamp   = -b * v
        m       =   5
        f0      =  40
        omega0  =   6
        k       = 500
        b       =  20
```

This function contains most of the physics of the particular situation we are considering. The following functions help calculate things that we calculated by hand above, so we can check that things are working as we expect.

```
ddHarmTuples :: [StateTuple1D]
ddHarmTuples = eulerSolution1D 0.1 accelFuncDampedDrivenHarm (0.0,0.1,0.0)

ddHarmTupleAtp3 :: StateTuple1D
ddHarmTupleAtp3 = ddHarmTuples !! 3

timeFromTuple :: StateTuple1D -> Time
timeFromTuple (t,_,_) = t

posFromTuple  :: StateTuple1D -> Time
posFromTuple  (_,x,_) = x

velFromTuple  :: StateTuple1D -> Time
velFromTuple  (_,_,v) = v
```

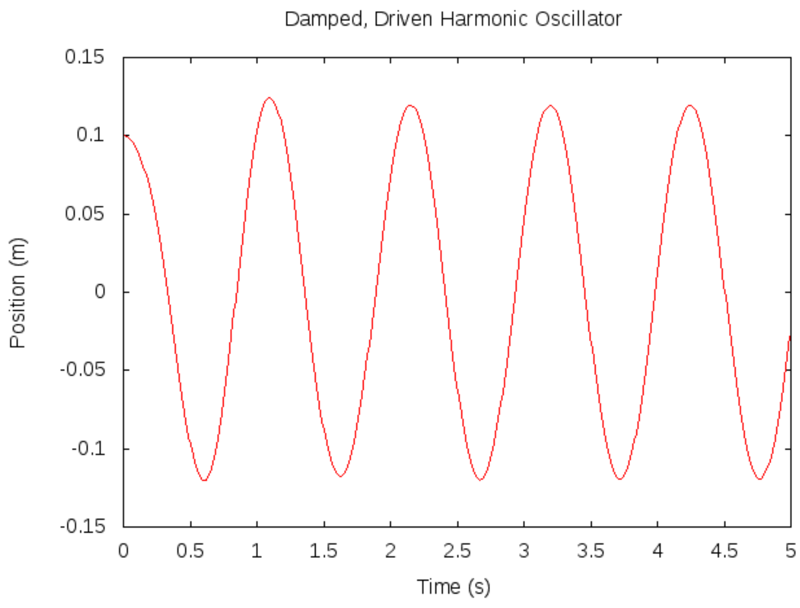
```
GHCi  :l Mechanics1D.lhs
GHCi  posFromTuple ddHarmTupleAtp3
          ⇝  3.402684919277425e-2
GHCi  velFromTuple ddHarmTupleAtp3
          ⇝  -0.7859527012620158
```

Before we leave the damped, driven harmonic oscillator, let's make a graph of its position as a function of time. I'm going to trim the time step to $\Delta t = 0.01$ s to get more accurate results.

```
ddHarmTuples' :: [StateTuple1D]
ddHarmTuples' = eulerSolution1D 0.01 accelFuncDampedDrivenHarm (0.0,0.1,0.0)

posTimePlot :: IO ()
posTimePlot
    = plotPath [Title "Damped, Driven Harmonic Oscillator"
```

```
            ,XLabel "Time (s)"
            ,YLabel "Position (m)"
            ,PNG "ddharm.png"
            ,Key Nothing
            ] [(t,x) | (t,x,_) <- take 500 ddHarmTuples']
```

Here is the plot we made.



13.2 Units

Another issue that we must consider is the choice of units. All of our physical quantities have units associated with them, but we are only asking the computer to keep track of numbers, and not units. In many problems, we can use the standard SI units (kilograms, meters, seconds, etc.) and in such a case we just remember that the numbers that are coming out of our calculations are in the standard SI units. But, suppose you are doing a calculation of the Earth's position as it revolves about the Sun. Meters might not be the best units to use, because you would always be dealing with numbers in the billions. For a problem like this, it makes sense to choose a bigger unit of length (maybe the astronomical unit AU), so that you can work with numbers that are within a few orders of magnitude of 1.

The same issues are true for microscopic problems in physics. If you are interested in the energy of an atom, it makes little sense to choose Joules as your unit of energy, because then you are always looking at tiny tiny numbers. Maybe electron volts (eV) or milli-electron volts (meV) would be better.

13.3 Air resistance

13.3.1 Introduction

When studying topics like projectile motion in an introductory physics class, we usually ignore air resistance.

Air resistance is the force an object feels as it moves through the air. It comes from the molecules of air slamming into the object.

In fact, air resistance is just one example of the larger subject of fluid flow. Air is a fluid (fluids are liquids and gases) that flows around an object. Fluids like air can exert forces on objects.

The study of fluid flow around solid objects is very complex. If a person wanted to design a good wing cross-section for an airplane, she would need to have a good understanding of this subject. Our aims are more modest. We don't really care what happens to the air as it flows around an object. We just want a decent model for the force that the air exerts on an object.

There is one thing that we can say right away. The direction of the force of air resistance on an object is opposite the motion of the object (in other words, opposite the velocity of the object).

13.3.2 Collision model

Let us think of the interaction between an object and the air around it as a collision. Suppose the object is moving with velocity v. Let the cross-sectional area of the object be A. Let the density of air be ρ.

We assume that the initial velocity of the air is zero, 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 vdt. The volume of air swept out by the object in time dt is $Avdt$. The mass of air disturbed by the object in time dt is

$$\rho A v dt.$$

The momentum imparted to the air by the object in time dt is

$$\rho A v^2 dt.$$

The force felt by the object from the air is

$$F_{\text{air}} = -\rho A v^2.$$

Our derivation was really quite approximate, since we don't know that the air molecules really end up with velocity v, and we haven't even tried to take into account 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, however, and 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 trying to fly through the air. Our final expression for the magnitude of air resistance is

$$F_{\text{air}} = -C\rho A v^2.$$

13.4 Euler-Cromer method

For second-order differential equations (such as Newton's second law), there is a slight modification that we can make to the Euler method which improves the results in a number of cases. Recall that we wish to solve the differential equation

$$\frac{d^2x}{dt^2} = \frac{1}{m} F_{\text{net}}\left(x, \frac{dx}{dt}, t\right). \tag{13.2}$$

By introducing $v = dx/dt$, this second-order differential equation is equivalent to the two first-order differential equations

$$\frac{dv}{dt} = \frac{1}{m} F_{\text{net}}(x, v, t) \tag{13.3}$$

$$\frac{dx}{dt} = v. \tag{13.4}$$

The Euler method prescribes that we find the values of x and v at a later time $t + \Delta t$ from their values at the earlier time t in the following way.

$$\begin{aligned} v(t + \Delta t) &= v(t) + \frac{1}{m} F_{\text{net}}(x(t), v(t), t)\Delta t \\ x(t + \Delta t) &= x(t) + v(t)\Delta t \end{aligned}$$

The Euler-Cromer method seeks a solution to the same differential equation(s), and the prescription is only slightly different.

$$v(t + \Delta t) = v(t) + \frac{1}{m} F_{\text{net}}(x(t), v(t), t)\Delta t \qquad (13.5)$$
$$x(t + \Delta t) = x(t) + v(t + \Delta t)\Delta t \qquad (13.6)$$

13.5 Exercises

Exercise 13.1. (Euler method by hand 2) Consider the differential equation

$$\frac{d^2x}{dt^2} = -3x + 4\cos 2t - 2\frac{dx}{dt}$$

for the function $x(t)$, along with the initial conditions

$$x(0) = 2$$

and

$$\frac{dx}{dt}(0) = 1.$$

Use the Euler method with a step size of $\Delta t = 0.1$ to approximate the value of $x(0.3)$.

Keep at least four figures after the decimal point in your calculations. Show your calculations in a small table. (The table will have three columns now, for t, x, and dx/dt.)

Exercise 13.2. (Euler method by computer 2) Write a Haskell function

```
eulerStep2 :: (Double,Double,Double) -> (Double,Double,Double)
eulerStep2 = undefined
```

that takes a pair (t, x, v) and returns a pair (t', x', v') for a single step of the Euler method for the differential equation

$$\frac{d^2x}{dt^2} = -3x + 4\cos 2t - 2\frac{dx}{dt}$$

with a step size of $\Delta t = 0.1$, and initial conditions as above.

Show how to use the function `eulerStep2` to calculate the value $x(0.3)$ that you calculated by hand before.

Exercise 13.3. Let's warm up with a basic projectile motion problem where we know what the answer should look like. Suppose a ball is thrown from the ground straight up into the air with an initial velocity of 10 m/s. Ignoring air resistance, use the Euler method to find the height of the ball as a function of time. Make a plot of height as a function of time.

Exercise 13.4. Consider a 3-kg mass connected by a linear spring with spring constant 100,000 N/m to a wall. If the spring is extended by 0.01 m and released, what does the subsequent motion look like? Investigate this motion over several cycles of oscillation. Compare your results to the exact solution. Find a time step that is small enough so that the Euler solution and the exact solution overlap precisely on a plot. Find another time step that is big enough so 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, etc.) 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". Put your name on the plot (using the label commands we learned).

Exercise 13.5. Let's investigate dropping things from large heights. In particular, let's look at a ping-pong ball and a bowling ball. In each case, take $C = 1/2$. You will need to find out good approximations for things like the size and mass of these balls. Let's drop them from 100 m and 500 m. Make graphs of velocity as a function of time and velocity as a function of vertical position. What fraction of terminal velocity is achieved in each case? Assemble your results in some meaningful and understandable way.

Exercise 13.6. In the Euler method, we update the state tuple $(t, x, v) \to$

(t', x', v') by

$$\begin{aligned} t' &= t + \Delta t \\ x' &= x + v\Delta t \\ v' &= v + \frac{1}{m} F_{\text{net}}(x, v, t)\Delta t \end{aligned}$$

Write similar update equations for the Euler-Cromer method.

Exercise 13.7. Return to the harmonic oscillator problem that we did earlier. Compare the Euler and Euler-Cromer solutions to the exact solution for a time step of 0.001 s (you will recall that this is not a very good time step for the Euler method). Plot the displacement of the mass as a function of time for the first 0.1 s of motion. Plot Euler, Euler-Cromer, and exact solutions on one set of axes. Also give the value of the position of the mass (to four significant figures) at $t = 0.1$ s for each of the three solutions.

Chapter 14

Mechanics in Three Dimensions

To do mechanics in three dimensions, we must use vectors rather than numbers for displacement, velocity, acceleration, and force.

14.1 Vectors in Haskell

Haskell does not come with a built-in type for vectors, so we have to define them ourselves. I have done this in the module `SimpleVec` in Appendix B. The module defines a new `Vec` type for vectors, along with a number of functions to work with vectors. Alternatively, the `Vec` type and related functions are also available in the `Physics.Learn.SimpleVec` module from the *learn-physics* package at `http://hackage.haskell.org/`. To use the module, we first import it.

GHCi
```
:m Physics.Learn.SimpleVec
```

You can make a vector with the `vec` function by giving its three components.

GHCi
```
vec 3.2 0 (-6.4)
    ⇝  vec 3.2 0.0 (-6.4)
```

What can we do with `Vec`s? We can add them to other `Vec`s and we can scale them with `Double`s. To add `Vec`s, we will not use the `+` operator that we use with numbers, but rather a new vector addition operator named `^+^`.

GHCi
```
vec 1 2 3 ^+^ vec 4 5 6
    ⇝  vec 5.0 7.0 9.0
```

You can think of the carrot on each side of the plus as a reminder that there is a vector on the left and a vector on the right. To scale a vector, we can use the `*^` operator.

GHCi
```
5 *^ vec 1 2 3
    ⇝  vec 5.0 10.0 15.0
```

Notice that the carrot goes on the right of the asterisk, because the vector is on the right. You can multiply a `Vec` by a `Double` on the right with the `^*` operator.

GHCi
```
vec 1 2 3 ^* 5
    ⇝  vec 5.0 10.0 15.0
```

Since the vector is on the left, the carrot is on the left. We can divide by a `Double` with the `^/` operator.

GHCi
```
vec 1 2 3 ^/ 5
    ⇝  vec 0.2 0.4 0.6
```

You can subtract `Vec`s with the `^-^` operator.

GHCi
```
vec 1 2 3 ^-^ vec 4 5 6
    ⇝  vec (-3.0) (-3.0) (-3.0)
```

You can take the dot product of two `Vec`s with the `<.>` operator.

GHCi
```
vec 1 2 3 <.> vec 4 5 6
    ⇝  32.0
```

You can take the cross product of two `Vec`s with the `><` operator (it's supposed to look like a cross product).

GHCi
```
vec 1 2 3 >< vec 4 5 6
    ⇝  vec (-3.0) 6.0 (-3.0)
```

You can take the magnitude of a vector.

GHCi
```
magnitude $ vec 3 4 12
    ⇝  13.0
```

If you need the components of a vector, you can get them with the `xComp` function.

GHCi
```
xComp $ vec 1 2 3 >< vec 4 5 6
    ⇝  -3.0
```

There are also functions `yComp` and `zComp`. The zero vector is called `zeroV`.

Function		Type
`zeroV`	`::`	`Vec`
`iHat`	`::`	`Vec`
`(^+^)`	`::`	`Vec -> Vec -> Vec`
`(^-^)`	`::`	`Vec -> Vec -> Vec`
`(*^)`	`::`	`Double -> Vec -> Vec`
`(^*)`	`::`	`Vec -> Double -> Vec`
`(^/)`	`::`	`Vec -> Double -> Vec`
`(<.>)`	`::`	`Vec -> Vec -> Double`
`(><)`	`::`	`Vec -> Vec -> Vec`
`negateV`	`::`	`Vec -> Vec`
`magnitude`	`::`	`Vec -> Double`
`xComp`	`::`	`Vec -> Double`
`vec`	`::`	`Double -> Double -> Double -> Vec`
`sumV`	`::`	`[Vec] -> Vec`

Table 14.1: Functions for working with vectors.

GHCi
```
zeroV
    ⇝  vec 0.0 0.0 0.0
```

You can negate a vector with `negateV`.

GHCi
```
negateV $ vec 1 2 3 >< vec 4 5 6
    ⇝  vec 3.0 (-6.0) 3.0
```

The unit vectors `iHat`, `jHat`, and `kHat` are defined.

GHCi
```
jHat
    ⇝  vec 0.0 1.0 0.0
```

Strictly speaking, `zeroV` and `iHat` are not functions. Is there a better word?

14.2 Euler method with vectors

Now that we have a vector type `Vec`, we need versions of our Euler method and Euler-Cromer method functions that work with vectors.

```
{-# OPTIONS_GHC -Wall #-}

import Physics.Learn.SimpleVec
import Graphics.Gloss
import Graphics.Gloss.Interface.Pure.Simulate
```

The very first line of code above turns on warnings, which is something I like to do because it encourages good programming habits. Later in this chapter we'll be using the Gloss animation library, so in addition to importing `Physics.Learn.SimpleVec`, we also import two Gloss modules.

For any code in which we want to use the `Vec` type and the associated operations, we must include

```
import Physics.Learn.SimpleVec
```

at the top of the file.

Suppose we define the following type synonyms.

```
type Time         = Double
type TimeStep     = Double
type Mass         = Double
type Position     = Vec
type Velocity     = Vec
type Acceleration = Vec
type Force        = Vec

type StateTuple = (Time,Position,Velocity)

type AccelerationFunction = StateTuple -> Acceleration
```

```
eulerStep :: TimeStep
          -> AccelerationFunction
          -> StateTuple -> StateTuple
eulerStep dt a (t,r,v) = (t + dt, r ^+^ dr, v ^+^ dv)
    where
```

```
      dr = v        ^* dt
      dv = a(t,r,v) ^* dt
```

```
eulerCromerStep :: TimeStep
                -> AccelerationFunction
                -> StateTuple -> StateTuple
eulerCromerStep dt a (t,r,v) = (t',r',v')
    where
      t' = t + dt
      r' = r ^+^ v' ^* dt
      v' = v ^+^ a(t,r,v) ^* dt
```

We are now in a wonderful position. All we need to do to solve any one-body problem in mechanics is give the computer

- a time step,
- the body's acceleration as a function of its current state,
- and the body's initial state.

Of course, to find the acceleration of the body, we will generally need to add up the forces that act on the body to get the net force, and divide by the body's mass. The computer will then calculate the position and velocity of the particle at later points in time, for as long as we like.

Let's make our technology even easier to use. Our function `eulerCromerStep` advances the scene by one time step, and we need to use the `iterate` function to move forward in time by many time steps. The function `eulerCromerSolution` returns an infinite list of `StateTuple`s, which can then be further processed for plotting or other purposes.

```
eulerCromerSolution :: TimeStep
                    -> AccelerationFunction
                    -> StateTuple -> [StateTuple]
eulerCromerSolution dt a = iterate $ eulerCromerStep dt a
```

Since Haskell is a lazy language, it will not actually calculate the entire infinite list (an impossible task in any case). It will only calculate the minimum amount of the list necessary to do what you ask of it.

Below is an example of how we might use our new Euler-Cromer vector technology for a simple projectile motion problem.

```
trajectory :: [StateTuple] -> [(Double,Double)]
trajectory tups = [(xComp r,yComp r) | (_,r,_) <- tups]

-- vertical direction is y direction
earthSurfaceGravity :: AccelerationFunction
earthSurfaceGravity _state = vec 0 (-g) 0

g :: Double
g = 9.81

projectileTuples :: Double -> Double -> [StateTuple]
projectileTuples v0 theta
    = eulerCromerSolution 0.01 earthSurfaceGravity
      (0,vec 0 0 0,vec vx0 vy0 0)
      where
        vx0 = v0 * cos theta
        vy0 = v0 * sin theta

plotTuples :: [StateTuple]
plotTuples = takeWhile (\(_,r,_) -> yComp r >= 0)
             $ projectileTuples 30 (pi/3)

plot1 :: IO ()
plot1 = plotPath [] $ trajectory plotTuples
```

Activity 14.1. Let us treat the Earth as being fixed at the origin of our coordinate system. Consider the gravitational force on a satellite of mass m, initial position $\mathbf{r}_0$, and initial velocity $\mathbf{v}_0$. Since the motion of the satellite will take place in a plane, we can use vectors that lie in the xy plane. Plot trajectories of orbits resulting from various initial conditions. Choose some values for initial conditions that give nearly circular orbits and some others that give elliptical orbits. You will find that the Euler method produces orbits that don't close on themselves. Hand in one plot comparing the Euler and Euler-Cromer methods for one orbit that you like (elliptical or circular). Indicate the step size that you used for the Euler and Euler-Cromer methods, as well as your choice of initial conditions.

Activity 14.2. The Lorentz force law describes the force exerted on a particle with charge q and velocity $\mathbf{v}$ by an electric field $\mathbf{E}$ and a magnetic field $\mathbf{B}$. The Lorentz force law is

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}).$$

Consider a uniform magnetic field in the z-direction. You may already know that a charged particle with initial velocity in the x-direction will go in circles in this magnetic field. Choose some values for the strength of the magnetic field, the charge of the particle, the mass of the particle, and the initial velocity. Confirm, using the Euler method, that the particle does indeed go in circles. Plot y vs. x for different time steps.

14.3 Animation

In section 11.1.2, we discussed the Gloss library's `animate` function. The key ingredient there was a function `Float -> Picture` that describes how a picture (the `Picture`) changes with time (the `Float`). The trouble with using the `animate` function to display the results of our calculations is that we don't have an explicit solution as a function of time, so we can't make a picture as a function of time.

Fortunately, the Gloss library provides another function, `simulate`, which works very nicely with numerical methods for solving differential equations (such as the Euler method), where we don't have a solution in advance.

The type of `simulate` is the following.

```
simulate :: Display   -- ^ Display mode.
         -> Color     -- ^ Background color.
         -> Int       -- ^ Number of simulation steps to take
                      --   for each second of real time.
         -> model     -- ^ The initial model.
         -> (model -> Picture)
                      -- ^ A function to convert
                      --   the model to a picture.
         -> (ViewPort -> Float -> model -> model)
                      -- ^ A function to step the model one
                      --   iteration. It is passed the
```

```
                        --   current viewport and the amount
                        --   of time for this simulation
                        --   step (in seconds).
         -> IO ()
```

The types `Display`, `Color`, and `Picture` are the same as for the `display` function we discussed in section 11.1.1.

The third argument to `simulate`, with type `Int`, is the number of simulation steps to take for each second of real time. I recommend starting with something in the range of 10–100. This "real time" is really *animation time* (time in the displayed simulation), which could be different from time in the physical situation you are working with. For example, if your physical situation is a satellite orbiting earth, it could take hours, days, or months for the satellite to make a revolution. In the animation, you may want to see a full revolution take place in 10 seconds. If you set the number of simulation steps per second to 20, you are asking for about 200 time steps per revolution if the revolution is displayed in 10 seconds.

The fourth argument to `simulate` has the type variable `model`. You can choose `model` to be any type you like. The type you choose should contain the full state of affairs necessary to describe your physical situation. For a single particle in three dimensions, a good choice for `model` is `StateTuple`. The fourth argument is an initial value of this `model`. This is the same information we need to give to the Euler or Euler-Cromer method.

The fifth argument to `simulate`, with type `model -> Picture`, is the heart of the animation. Here you need to write a function that describes what the `Picture` should look like for a given value of type `model`. (Perhaps "state" would have been a better type variable than "model".) If you are using `StateTuple` as your `model`, the function needs to describe how you will use the time, position, and velocity contained in the `StateTuple` to construct a picture. You do not need to use all three. Perhaps you wish only to display information about the position. It's up to you.

The sixth argument to `simulate`, with type

`ViewPort -> Float -> model -> model`,

describes how the system evolves in time. This is closely related to our `eulerCromerStep` function, and should probably be based on it. Do not worry about the `ViewPort` type. You can give a dummy variable for `ViewPort` and not use it in your function (the underscore _ is Haskell's conventional name for a dummy argument that you have no intention of using. The Float is a time step. Note that eulerCromerStep also takes a time step as one of its arguments. If you need to scale the time (from a day of time in your physical situation to a second of time in animation, for example), this a good place to do it. The Float input to this function is a time step in seconds of *animation time*. You may want to scale this time before passing it as a time step to eulerCromerStep.

Example 14.1. Here is a full working example of Gloss's simulate function being used to animate projectile motion.

```
{-# OPTIONS_GHC -Wall #-}

-- Use Gloss's simulate function to animate projectile motion
-- using the Euler-Cromer method

-- positive x is to the right in Translate
-- positive y is up           in Translate (this is good)

-- This function defines a disk
-- in terms of Gloss's ThickCircle
disk :: Float -> Picture
disk radius = ThickCircle (radius/2) radius

-- A red disk will represent the projectile
redDisk :: Picture
redDisk = Color red (disk 50)

-- projectile mass
projectileMass :: Double
projectileMass = 1

-- initial time
t0 :: Double
t0 = 0

-- initial position
```

```haskell
r0 :: Vec
r0 = vec 0 0 0

-- given a speed and angle, produce a velocity vector
velFromSpeedAngle :: Double -> Double -> Vec
velFromSpeedAngle v theta
    = vec (v * cos theta) (v * sin theta) 0

-- initial velocity
v0 :: Vec
v0 = velFromSpeedAngle 100 (35 * pi / 180)

-- Gloss's idea of "model" is the same as our idea of state,
-- that is, the information required to express the current
-- "state of affairs" of our system.
-- The initial model is the initial StateTuple.
modelInitial :: StateTuple
modelInitial = (t0,r0,v0)

-- This function tells what the picture should look like
-- for a given state of affairs.
modelToPicture :: StateTuple -> Picture
modelToPicture (_t,r,_v)
    = scale 0.2 0.2 $ translate xFloat yFloat redDisk
      where
        xFloat = realToFrac (xComp r)
        yFloat = realToFrac (yComp r)

-- By default, gloss will attempt to do the simulation
-- in real time.  If we're animating the orbit of the moon
-- around the Earth, we don't want to watch the computer
-- for one month to see one orbit.
-- A number greater than one here is a speedup factor.
-- A number less than one is a slow-down factor.
timeScale :: Double
timeScale = 3
```

```
projectileMotionNetForce :: Mass -> Force
projectileMotionNetForce m = m *^ vec 0 (-9.8) 0

projectileMotionAccelerationFunction :: Mass
                                     -> AccelerationFunction
projectileMotionAccelerationFunction m (_t,_r,_v)
    = projectileMotionNetForce m ^/ m

-- The rule for updating the "model" is just to take
-- one Euler step!
simStep :: ViewPort -> Float -> StateTuple -> StateTuple
simStep _ dt
    = eulerCromerStep dtScaled
      (projectileMotionAccelerationFunction projectileMass)
          where
            dtScaled = timeScale * realToFrac dt

-- The main program creates a window and does our animation.
-- We need to give it an initial "model" (modelInitial),
-- a function to translate from model to picture
-- (modelToPicture), and an update rule (simStep).
main :: IO ()
main = simulate
       (InWindow "Projectile Motion" (600, 600) (10, 10))
       white 20 modelInitial modelToPicture simStep
```

Activity 14.3. Return to the satellite orbiting the Earth. Write a Haskell program to animate your satellite's motion around the Earth. Show that by using different initial conditions, you can achieve circular orbits and elliptical orbits.

Chapter 15

Multiple Objects in Three Dimensions

15.1 The State of a Physical System

One fruitful way to structure our thinking about a number of physical theories revolves around the concept of *state*. The state-based paradigm picks out time as a special quantity. The state of a physical system is the collection of information needed to say precisely what is going on with the system *at this instant of time*. The state represents the current "state of affairs" of the system. At a later instant of time, we expect the system to be described by a different state.

Given a physical system that we wish to understand (say the motion of a baseball), the state-based paradigm suggests the following conceptual division.

1. Exactly 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?

Question 1 asks how we will model our system. If our system is a baseball, one choice is to model the system as a point particle, ignoring size and rotation of the ball. In this case, we could choose the state to consist of the instantaneous displacement of the ball from some reference point along with the instantaneous velocity of the ball. Alternatively, if the rotation of the ball is important, we could model the ball as a rigid body. In this case, the state would consist of the displacement and velocity of the center of the ball along with parameters describing the orientation and angular velocity of the ball. A third alternative, if we wish to take into consideration air resistance on a windy day, could require parameters to describe the velocity and even perhaps the density of the air through which the ball passes.

Question 3 requires a physical theory to answer. In the case of mechanics, Newton's second law (along with Newton's third law for multiple objects) gives the rule, in the form of a differential equation, by which the state changes in time.

The quantities that we are often interested in are functions of the system state.

Question 2 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 1 is a data type. The answer to question 2 is a value of this data type.

Let's illustrate these ideas by writing some code to animate the Sun, Earth, and Moon evolving under their mutual gravitational attraction.

```haskell
{-# OPTIONS_GHC -Wall #-}

-- Animation of Earth and Moon orbiting around a fixed Sun

import Physics.Learn.SimpleVec
import Graphics.Gloss
import Graphics.Gloss.Interface.Pure.Simulate

type Time         = Double
type TimeStep     = Double
type Pos          = Vec
type Vel          = Vec
type Acceleration = Vec

type Mass         = Double
type GravConstant = Double

-- First Earth, then Moon
type SystemState = (Time,[(Pos,Vel)])

type AccelerationFunction = SystemState -> [Acceleration]

-- SEM units

-- Let's depart from SI units and use units chosen for the problem.
-- This way, our numbers will be closer to 1 and easier to think about.

-- 1 SEMLength = Sun-Earth separation
-- 1 SEMMass   = 1 Solar mass (Sun's mass)
-- 1 SEMTime   = 1/12 Year

lengthFactor, massFactor, timeFactor :: Double
lengthFactor = 1 / 1.496e11    -- SEMLength / m
massFactor   = 1 / 1.99e30     -- SEMMass / kg
timeFactor   = 12 / (365.25 * 24 * 60 * 60)   -- SEMTime / s

gGrav :: GravConstant
gGrav = 6.67e-11 * lengthFactor**3 / massFactor / timeFactor**2

massSun, massEarth, massMoon :: Mass
massSun   = 1.99e30 * massFactor
massEarth = 5.98e24 * massFactor
massMoon  = 7.35e22 * massFactor

radiusSun :: Double
--radiusSun = 6.96e8 * lengthFactor
radiusSun = 0.25

radiusEarth :: Double
--radiusEarth = 6.38e6 * lengthFactor
radiusEarth = 0.04
```

```
radiusMoon :: Double
--radiusMoon = 1.74e6 * lengthFactor
radiusMoon = 0.02

earthSunDistance :: Double
earthSunDistance = 1.496e11 * lengthFactor

year :: Double
year = 365.25*24*60*60 * timeFactor

moonEarthDistance :: Double
moonEarthDistance = 3.84e8 * lengthFactor

-- Derived constants

initialEarthSpeed :: Double
initialEarthSpeed = 2*pi*earthSunDistance/year

initialMoonSpeedwrtE :: Double
initialMoonSpeedwrtE = sqrt (gGrav * massEarth / moonEarthDistance)

initialState1 :: SystemState
initialState1 = (0,[(vec earthSunDistance 0 0,vec 0 initialEarthSpeed 0)
                   ,(vec (earthSunDistance + moonEarthDistance) 0 0
                    ,vec 0 (initialEarthSpeed + initialMoonSpeedwrtE) 0)])

eulerCromerStep :: TimeStep -> AccelerationFunction -> SystemState -> SystemSt
eulerCromerStep dt a (t,rvs) = (t + dt,rvsNew)
    where
      as = a (t,rvs)
      (rs,vs) = unzip rvs
      rsNew = zipWith (^+^) rs (map (^* dt) vsNew)
      vsNew = zipWith (^+^) vs (map (^* dt) as)
      rvsNew = zip rsNew vsNew

earthMoonGravity :: AccelerationFunction
earthMoonGravity (_,[(rE,_),(rM,_)]) = [aE,aM]
```

```
    where
      rEM = rE ^-^ rM
      fES = (-gGrav) * massEarth * massSun  *^ rE  ^/ magnitude rE  ** 3
      fEM = (-gGrav) * massEarth * massMoon *^ rEM ^/ magnitude rEM ** 3
      fE  = fES ^+^ fEM
      aE  = fE ^/ massEarth
      fMS = (-gGrav) * massMoon  * massSun  *^ rM  ^/ magnitude rM  ** 3
      fME = negateV fEM
      fM  = fMS ^+^ fME
      aM  = fM ^/ massMoon
earthMoonGravity _ = error "earthMoonGravity: bad error"

fakeMoonPos :: Vec -> Vec -> Vec
fakeMoonPos rE rM = rE ^+^ 50 *^ (rM ^-^ rE)

-- Gloss's idea of "world" is the same as our idea of state,
-- that is, the information required to express the current "state of affairs"
-- of our system.
-- The initial world is the initial SystemState.
worldInitial :: SystemState
worldInitial = initialState1

-- This function defines a disk in terms of Gloss's ThickCircle
disk :: Double -> Picture
disk r = ThickCircle (radius/2) radius
    where radius = realToFrac r

-- A yellow disk will represent the Sun
yellowDisk :: Picture
yellowDisk = Color yellow (disk radiusSun)

-- A blue disk will represent the Earth
blueDisk :: Picture
blueDisk = Color blue (disk radiusEarth)

-- A white disk will represent the Moon
whiteDisk :: Picture
whiteDisk = Color white (disk radiusMoon)

-- This function tells what the picture should look like for a given
-- state of affairs.
worldToPicture :: SystemState -> Picture
worldToPicture (_t,[(rE,_vE),(rM,_vM)])
    = scale 200 200 $ pictures [yellowDisk
                               ,translate xE yE blueDisk
                               ,translate xMf yMf whiteDisk
                               ]
    where
      xE = realToFrac (xComp rE)
      yE = realToFrac (yComp rE)
      xMf = realToFrac (xComp $ fakeMoonPos rE rM)
      yMf = realToFrac (yComp $ fakeMoonPos rE rM)
worldToPicture _ = error "worldToPicture:  unexpected input"

-- By default, gloss will attempt to do the simulation in real time.
-- A number greater than one here is a speedup factor.
-- A number less than one is a slow-down factor.
-- A time of one means one month (not one second, although the computer
-- will interpret it as one second).
timeScale :: Double
timeScale = 1

-- The rule for updating the "world" is just to take one Euler step!
simStep :: ViewPort -> Float -> SystemState -> SystemState
simStep _ dt = eulerCromerStep dtScaled earthMoonGravity
    where
      dtScaled = timeScale * realToFrac dt

-- The main program creates a window and does our animation.
-- We need to give it an initial "world" (worldInitial),
-- a function to translate from world to picture (worldToPicture),
-- and an update rule (simStep).
main :: IO ()
main = simulate (InWindow "Sun, Earth, Moon Animation" (600, 600) (10, 10))
```

```
        black 20 worldInitial worldToPicture simStep
```

15.2 Multiple Objects in Three Dimensions

If we are interested in the motion of more than one particle in three dimensions, we need to keep track of the position and velocity for *each* particle.

Download the file `sunEarthMoonTemplate.hs` with the following command.

```
wget http://quantum.lvc.edu/walck/phy261/sunEarthMoonTemplate.hs
```

Take a look at this code to get an idea for how we might represent multiple objects in Haskell. The code has some `undefined` functions that need to be filled in before it will run.

Activity 15.1. Using realistic initial conditions, program an animation for the sun, earth, and moon mutually interacting though gravity. Fix the sun at the origin. Take into account the gravitational force of the moon on the earth, the earth on the moon, the sun on the earth, and the sun on the moon. The actual earth-sun separation is about 500 times the earth-moon separation, so you won't be able to resolve the earth and moon as separate objects on the screen. In order to be able to see where the moon is relative to the earth, I suggest the following.

Instead of displaying the moon at the position you calculate, display the moon at a "fake" position that has the correct orientation, but is 50 times as far from the Earth as you calculate. Here is an equation to use to calculate a fake moon position:

$$\mathbf{r}_{\text{FM}} = \mathbf{r}_{\text{E}} + A(\mathbf{r}_{\text{M}} - \mathbf{r}_{\text{E}})$$

where $\mathbf{r}_{\text{FM}}$ is the position of the fake moon, $\mathbf{r}_{\text{M}}$ is the position of the (real) moon, $\mathbf{r}_{\text{E}}$ is the position of the earth, and A is a magnification factor that artificially magnifies the vector from earth to moon for display purposes. Try $A = 50$ and see what happens. This fake moon is only for display purposes. Its position should not show up in any of the actual physics equations. Show me that your program works, and place a copy in my drop box.

15.3 Waves on a Flexible String

Imagine a flexible string, like a rubber band or a guitar string, with some tension in it. The ends of the string are located at $x = 0$ and $x = L$, and are held fixed at $y = 0$. The rest of the string is allowed to move.

We will view this string as being made up of $N + 1$ little pieces of mass, with neighboring pieces connected by a linear spring with spring constant k. The force on the jth little piece of mass is composed of two parts. There is a force from the neighbor on the left and a force from the neighbor on the right. The vector sum of these two forces is

$$\mathbf{F}_j = -k(\mathbf{r}_j - \mathbf{r}_{j-1}) - k(\mathbf{r}_j - \mathbf{r}_{j+1}).$$

We are going to ignore any effects of gravity in this string vibration. (We imagine that the elastic forces are much larger that the force of gravity would be.) The expression above is appropriate when $1 \leq j \leq N - 1$, that is, for all of the little pieces of mass except the ones on each end. The pieces of mass on the ends are held fixed. In other words, there is some other force present (from whatever is constraining the end of the string to stay fixed) so that the net force on the pieces of mass at the ends is zero.

$$\begin{aligned} \mathbf{F}_0 &= 0 \\ \mathbf{F}_N &= 0 \end{aligned}$$

Putting all of this together, the net force on the jth piece of mass is given by the following expression.

$$\mathbf{F}_j = \begin{cases} 0 & , \quad j = 0 \text{ or } j = N \\ -k(\mathbf{r}_j - \mathbf{r}_{j-1}) - k(\mathbf{r}_j - \mathbf{r}_{j+1}) & , \quad 1 \leq j \leq N - 1 \end{cases}$$

How do we talk about the initial conditions (or initial state) of the string? There are a number of ways we could do this. In general, the string could have both initial stretching (position) as well as initial velocity. Let's confine our attention to situations in which the initial velocity is zero. (So that we imagine pulling the string back into a certain shape and letting go of it.) One way to describe the initial configuration of the string is to give the vertical (y) position as a function of the horizontal (x) position. This is the method we use in the activity below.

Activity 15.2. Animate a wave on a string. Use the force equation given above to determine the motion of each little piece of the string. Try a number of initial conditions on the string to make different kinds of waves.

1. Use the initial conditions

$$\begin{aligned} y(x) &= y_0 \sin\frac{\pi x}{L} \\ \mathbf{v}(x) &= 0. \end{aligned}$$

   What would you call this wave?

2. Use the initial condition

$$\begin{aligned} y(x) &= y_0 \sin\frac{2\pi x}{L} \\ \mathbf{v}(x) &= 0. \end{aligned}$$

   What would you call this wave?

3. Use the initial condition

$$\begin{aligned} y(x) &= \begin{cases} y_0 \sin\frac{4\pi x}{L} & , \quad 0 \le x \le L/4 \\ 0 & , \quad x > L/4 \end{cases} \\ \mathbf{v}(x) &= 0. \end{aligned}$$

   How would you describe this wave in words?

You may choose whatever values you like for the constants that show up in this problem (k, L, y_0, etc.).

To get started, you may download the file `WaveOnStringTemplate.hs` with the following command and use it as a template for your program.

```
wget http://quantum.lvc.edu/walck/phy261/WaveOnStringTemplate.hs
```

Show me when you have your string animation working, and put a copy in my drop box. The program you submit should include all three initial conditions, with two of them commented out so that only one is “active”, but that either of the other two could be easily substituted to be the active initial condition.

Part III

Electromagnetic Theory

The great advance in the classical electromagnetic theory developed in the 19th century comes from the introduction of *fields*. Whereas Coulomb's late 18th century view of electrical phenomena relies on a force exerted directly by one charged object onto a distant charged object, the 19th century view has charges producing electric and magnetic fields and those fields exerting forces on charges. The electric and magnetic fields become players on the theoretical stage of equal stature to the charged particles themselves, capable of carrying energy and momentum. The Maxwell equations describe how charges create fields, while the Lorentz force law describes how fields exert force on charges.

Include Maxwell Equation/Lorentz force figure

Chapter 16

Coordinate Systems

Most of electromagnetic theory takes place in three-dimensional space. It will be very helpful to us to be able to use cylindrical coordinates and spherical coordinates in our description of three-dimensional space. Nevertheless, it is also helpful to begin with two-dimensional polar coordinates as a stepping stone to three-dimensional cylindrical coordinates.

16.1 Polar coordinates

We will use the variables s and ϕ for polar coordinates. The coordinate s is the distance from the origin to a point in the plane, and the coordinate ϕ is the angle between the x axis and a line joining the origin to a point. See Figure 16.1.

The Cartesian coordinates x and y are related to the polar coordinates s and ϕ by the following equations.

$$\begin{aligned} x &= s \cos \phi \\ y &= s \sin \phi \end{aligned}$$

We introduce polar coordinate unit vectors. The unit vector $\hat{\mathbf{s}}$ points away from the origin. (This is a well-defined direction at every point in the plane except for the origin itself.) Equivalently, the unit vector $\hat{\mathbf{s}}$ points in the direction for which ϕ stays constant and s increases. Similarly, the unit vector $\hat{\boldsymbol{\phi}}$ points in the direction for which s stays constant and ϕ increases. We can write the polar coordinate unit vectors $\hat{\mathbf{s}}$ and $\hat{\boldsymbol{\phi}}$ in terms of the

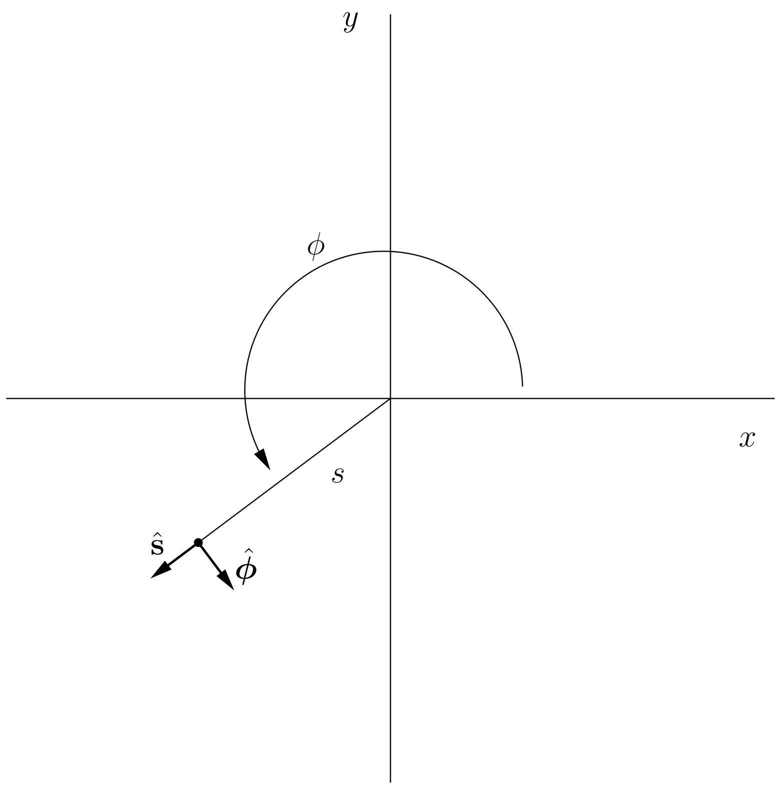


Figure 16.1: Polar coordinates.

Cartesian coordinate unit vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ as follows.

$$\hat{\mathbf{s}} = \frac{x\hat{\mathbf{x}} + y\hat{\mathbf{y}}}{\sqrt{x^2+y^2}} = \cos\phi\hat{\mathbf{x}} + \sin\phi\hat{\mathbf{y}}$$

$$\hat{\boldsymbol{\phi}} = -\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}}$$

Problem 16.1. Show that the polar coordinate unit vectors form an orthonormal system. In other words, show that

$$\begin{aligned} \hat{\mathbf{s}} \cdot \hat{\mathbf{s}} &= 1 \\ \hat{\boldsymbol{\phi}} \cdot \hat{\boldsymbol{\phi}} &= 1 \\ \hat{\mathbf{s}} \cdot \hat{\boldsymbol{\phi}} &= 0. \end{aligned}$$

Problem 16.2. Write $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ in terms of $\hat{\mathbf{s}}$ and $\hat{\boldsymbol{\phi}}$.

Unlike the Cartesian unit vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$, the polar unit vectors $\hat{\mathbf{s}}$ and $\hat{\boldsymbol{\phi}}$ point in different directions at different points in the plane.

16.2 Cylindrical coordinates

We can use the cylindrical coordinates s, ϕ, and z to represent the location of a point in space. The coordinate s is the distance from the z axis to the point in space, the coordinate ϕ is the angle between the xz plane and the plane containing the z axis and the point, and the coordinate z means the same thing as in Cartesian coordinates (the distance from the xy plane). See Figure 16.2. Cylindrical coordinates are closely related to polar coordinates in that cylindrical coordinates describe the xy plane in a polar fashion, but continue to use the Cartesian z coordinate.

The Cartesian coordinates x, y, and z are related to the cylindrical coordinates s, ϕ, and z by the following equations.

$$\begin{aligned} x &= s\cos\phi \\ y &= s\sin\phi \\ z &= z \end{aligned}$$

We introduce cylindrical coordinate unit vectors. The unit vector $\hat{\mathbf{s}}$ points away from the z axis. (This is a well-defined direction at every point in space

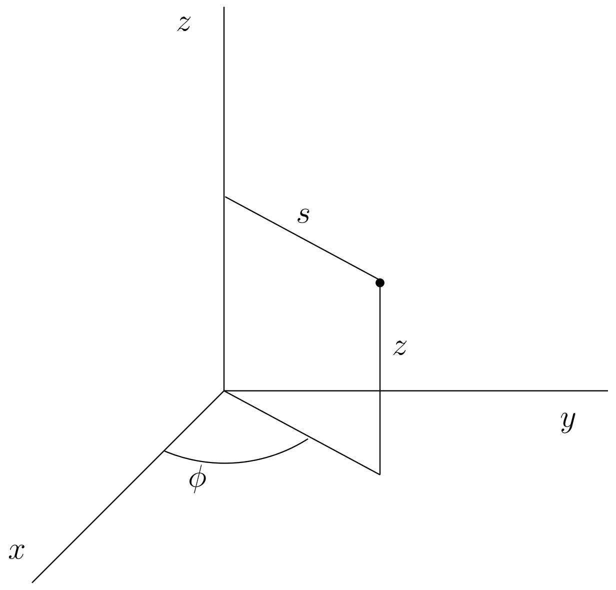


Figure 16.2: Cylindrical coordinates.

except for points on the z axis.) Equivalently, the unit vector $\hat{\mathbf{s}}$ points in the direction for which ϕ and z stay constant and s increases. Similarly, the unit vector $\hat{\boldsymbol{\phi}}$ points in the direction for which s and z stay constant and ϕ increases. Finally, the unit vector $\hat{\mathbf{z}}$ points in the direction for which s and ϕ stay constant and z increases. We can write the cylindrical coordinate unit vectors $\hat{\mathbf{s}}$, $\hat{\boldsymbol{\phi}}$, and $\hat{\mathbf{z}}$ in terms of the Cartesian coordinate unit vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ as follows.

$$\hat{\mathbf{s}} = \cos\phi\hat{\mathbf{x}} + \sin\phi\hat{\mathbf{y}}$$
$$\hat{\boldsymbol{\phi}} = -\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}}$$
$$\hat{\mathbf{z}} = \hat{\mathbf{z}}$$

16.3 Spherical coordinates

We can use the spherical coordinates r, θ, and ϕ to represent the location of a point in space. The coordinate r is the distance from the origin to the point in space, the coordinate θ is the angle between the z axis and a line from the origin to the point, and the coordinate ϕ is the angle between the

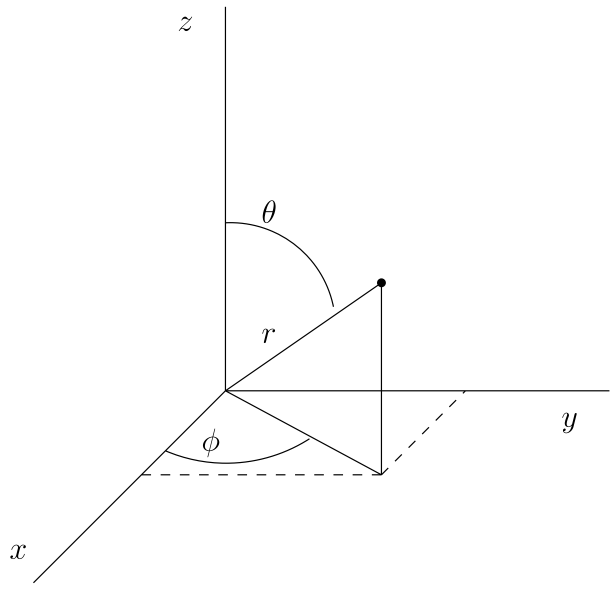


Figure 16.3: Spherical coordinates.

xz plane and the plane containing the z axis and the point. (The coordinate ϕ has the same meaning in spherical coordinates that it does in cylindrical coordinates.) See Figure 16.3.

The Cartesian coordinates x, y, and z are related to the spherical coordinates r, θ, and ϕ by the following equations.

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned}$$

We introduce spherical coordinate unit vectors. The unit vector $\hat{\mathbf{r}}$ points away from the origin. (This is a well-defined direction at every point in space except for the origin itself.) Equivalently, the unit vector $\hat{\mathbf{r}}$ points in the direction for which θ and ϕ stay constant and r increases. Similarly, the unit vector $\hat{\boldsymbol{\theta}}$ points in the direction for which r and ϕ stay constant and θ increases. Finally, the unit vector $\hat{\boldsymbol{\phi}}$ points in the direction for which r and θ stay constant and ϕ increases. To write $\hat{\mathbf{r}}$ in terms of the Cartesian unit vectors, we divide the position vector $\mathbf{r} = x\hat{\mathbf{x}} + y\hat{\mathbf{y}} + z\hat{\mathbf{z}}$ by its magnitude $r = \sqrt{x^2 + y^2 + z^2}$. The expression for $\hat{\boldsymbol{\phi}}$ is the same as it was for cylindrical coordinates. An expression for $\hat{\boldsymbol{\theta}}$ can be found from $\hat{\boldsymbol{\theta}} = \hat{\boldsymbol{\phi}} \times \hat{\mathbf{r}}$. We can write the spherical coordinate unit vectors $\hat{\mathbf{r}}$, $\hat{\boldsymbol{\theta}}$, and $\hat{\boldsymbol{\phi}}$ in terms of the Cartesian coordinate unit vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ as follows.

$$\begin{aligned}
\hat{\mathbf{r}} &= \frac{x\hat{\mathbf{x}} + y\hat{\mathbf{y}} + z\hat{\mathbf{z}}}{\sqrt{x^2+y^2+z^2}} = \sin\theta\cos\phi\hat{\mathbf{x}} + \sin\theta\sin\phi\hat{\mathbf{y}} + \cos\theta\hat{\mathbf{z}} \\
\hat{\boldsymbol{\theta}} &= \cos\theta\cos\phi\hat{\mathbf{x}} + \cos\theta\sin\phi\hat{\mathbf{y}} - \sin\theta\hat{\mathbf{z}} \\
\hat{\boldsymbol{\phi}} &= -\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}}
\end{aligned}$$

Problem 16.3. Show that the spherical coordinate unit vectors form an orthonormal system. In other words, show that

$$\begin{aligned}
\hat{\mathbf{r}} \cdot \hat{\mathbf{r}} &= 1 \\
\hat{\boldsymbol{\theta}} \cdot \hat{\boldsymbol{\theta}} &= 1 \\
\hat{\boldsymbol{\phi}} \cdot \hat{\boldsymbol{\phi}} &= 1 \\
\hat{\mathbf{r}} \cdot \hat{\boldsymbol{\theta}} &= 0 \\
\hat{\mathbf{r}} \cdot \hat{\boldsymbol{\phi}} &= 0 \\
\hat{\boldsymbol{\theta}} \cdot \hat{\boldsymbol{\phi}} &= 0.
\end{aligned}$$

Problem 16.4. Write $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ in terms of $\hat{\mathbf{r}}$, $\hat{\boldsymbol{\theta}}$, and $\hat{\boldsymbol{\phi}}$.

16.4 A type for position

We would like to have a Haskell type to describe the position of a point in space. We would further like to be able to specify points in 3D space in Cartesian, cylindrical, or spherical coordinates, and to access previously defined positions in any of the coordinate systems, including a system different from the one used to define it. Let's code.

We begin by setting warnings on. I like to do this to get feedback on things I might not have noticed and might not have intended.

```
{-# OPTIONS_GHC -Wall #-}
```

The code we define in this chapter will form a module. Let's give our module the name `CoordinateSystems` so that we can import the definitions we write here into our future work if we want to.

```haskell
module CoordinateSystems where
```

Here we import some definitions from another module. We need to do this at the top of the Haskell file, but we will explain the use of these imported types and functions later in the chapter.

```haskell
import SimpleVec
    ( Vec
    , vec
    , xComp
    , yComp
    , zComp
    , iHat
    , jHat
    , kHat
    , magnitude
    , (^/)
    , sumV
    )
```

I like to imagine that I am working with real numbers (even though they are approximations to real numbers), so I define a type `R` for real numbers. We use the keyword `type` to make a *type synonym*.

```haskell
type R = Double
```

The compiler now treats `R` and `Double` interchangeably. The type `R` is just a shorthand for `Double`. I do this just because I like looking at `R` more than I like looking at `Double`.

How can we use Haskell to describe a point in space? Option A is to use a triple `(R,R,R)` of Cartesian coordinates. This is fine for many purposes. It has the advantage of simplicity. It has the disadvantage that we already know we are interested in using cylindrical and spherical coordinates, which are also triples of numbers. This puts us in the dangerous position of mistaking a Cartesian (x, y, z) triple for a spherical (r, θ, ϕ) triple. The compiler can help us not to make this mistake, but only if we make intelligent use of the type system. Option A is workable, but dangerous. We can make better use of the computer to help us avoid mistakes.

Option B is to use the `Vec` type for position, as we did in mechanics. The `Vec` type clearly has Cartesian components, so it's harder to get confused compared with Option A. If we run into a triple `(R,R,R)` somewhere in code we've previously written, the type does not tell us whether it's a Cartesian triple or a spherical triple. On the other hand, if we run into a `Vec`, we know it is a Cartesian triple under the hood. Option B is workable. One downside of Option B is that position is not really a vector, because vectors are by definition things that can be added, and it doesn't make sense to add positions. If we think of position as a vector, it is vector from some fixed origin. But adding vectors means putting them tip-to-tail, and this isn't really allowed for position "vectors" whose tails are fixed at the origin. The other disadvantage of using `Vec` for position (Option B) is that the Haskell type system cannot help us to distinguish position from any other `Vec` (such as velocity, acceleration, or momentum).

Option C is to use Haskell's facilities to make a brand new data type ourselves, which can not be confused with any other data type. This is not the simplest option, but it will give us the power of working with the three coordinates systems we are interested in, and it will give us the advantage that the compiler will not allow us to confuse position with velocity. We will pursue Option C.

We construct a new type in Haskell with the `data` keyword.

```haskell
data Position = Cart R R R
                deriving (Show)
```

The `Position` that appears immediately to the right of the `data` keyword is the name we give to the new type. The `Cart` that appears to the right of the equals sign is called a *data constructor*. The type (`Position` in this case) and the data constructor (`Cart` in this case) can have the same name or different names. They live in different *namespaces*, so that even if they have the same name (they don't in this case), they are not the same object. (Which is why we put the type in blue and the data constructor in black.)

The data constructor is named `Cart` to indicate our intent to store positions in Cartesian coordinates regardless of the coordinate system used to describe the position.

The data type `Position` by itself is not terribly useful. It is just a new way to store three numbers. To be fair, it useful to have a way to store three numbers that the compiler will not confuse with any other way of storing three numbers (like a `Vec`). But the real usefulness of `Position` is that we will now define three ways of *making* a `Position` (one for each coordinate system), and three ways of *using* a `Position` (again, one for each coordinate system.

A coordinate system is a function from three real numbers to space.

```
type CoordinateSystem = (R,R,R) -> Position
```

Here are the definitions for the three coordinate systems. For Cartesian coordinates, we just stick the coordinates behind the data constructor `Cart`. For cylindrical coordinates (s, ϕ, z), we convert to Cartesian and then apply the `Cart` constructor to the Cartesian values. For spherical coordinates (r, θ, ϕ), we again apply the data constructor to the converted Cartesian values.

```
cartesian   :: CoordinateSystem
cartesian (x,y,z)
  = Cart x y z

cylindrical :: CoordinateSystem
cylindrical (s,phi,z)
  = Cart (s * cos phi) (s * sin phi) z

spherical   :: CoordinateSystem
spherical (r,theta,phi)
  = Cart (r * sin theta * cos phi)
         (r * sin theta * sin phi)
         (r * cos theta)
```

The functions `cartesian`, `cylindrical`, and `spherical` are our three ways of making a `Position`. Before we turn to the three ways of using a

`Position`, we'll define three helper functions that are almost the same as `cartesian`, `cylindrical`, and `spherical`. These three functions have the shortened names `cart`, `cyl`, and `sph`, and the only difference is that they take their arguments in a curried style, one right after the other, rather than in a triple. They are just convenient helping functions, and not really necessary.

```
cart :: R  -- ^ x coordinate
     -> R  -- ^ y coordinate
     -> R  -- ^ z coordinate
     -> Position
cart = Cart

cyl  :: R  -- ^ s coordinate
     -> R  -- ^ phi coordinate
     -> R  -- ^ z coordinate
     -> Position
cyl s phi z = cylindrical (s,phi,z)

sph  :: R  -- ^ r coordinate
     -> R  -- ^ theta coordinate
     -> R  -- ^ phi coordinate
     -> Position
sph r theta phi = spherical (r,theta,phi)
```

The function `cart` is a helping function to take three numbers x, y, and z and form the appropriate position using Cartesian coordinates. The definition of `cart` is given in *point-free style*. This means we omitted the parameters because they are identical on both sides of the equation. We could have written `cart x y z = Cart x y z` in the last line and it would mean the same thing. You can use regular style or point-free style, which ever you prefer. We also could have written `cart x y z = cartesian (x,y,z)`, which would follow the pattern in the cylindrical and spherical cases.

The function `cyl` is a helping function to take three numbers s, ϕ, and z and form the appropriate position using cylindrical coordinates. We just call the function `cylindrical` to do the real work. The function `sph` is a helping function to take three numbers r, θ, and ϕ and form the appropriate position using spherical coordinates.

We said earlier that we would like to be able to look at an existing Position in Cartesian, cylindrical, or spherical coordinates, regardless of the coordinate system used to define the position. The follow three functions show how to *use* a position to obtain a triple in the desired coordinate system.

```
cartesianCoordinates   :: Position -> (R,R,R)
cartesianCoordinates   (Cart x y z) = (x,y,z)

cylindricalCoordinates :: Position -> (R,R,R)
cylindricalCoordinates (Cart x y z) = (s,phi,z)
    where
      s = sqrt(x**2 + y**2)
      phi = atan2 y x

sphericalCoordinates   :: Position -> (R,R,R)
sphericalCoordinates   (Cart x y z) = (r,theta,phi)
    where
      r = sqrt(x**2 + y**2 + z**2)
      theta = atan2 s z
      s = sqrt(x**2 + y**2)
      phi = atan2 y x
```

16.5 Displacement

A *displacement* is a vector. It is the vector that points from a source position to a target position.

```
type Displacement = Vec

displacement :: Position  -- ^ source position
             -> Position  -- ^ target position
             -> Displacement
displacement (Cart x' y' z') (Cart x y z) = vec (x-x') (y-y') (z-z')

-- | Shift a position by a displacement.
```

```
shiftPosition :: Displacement -> Position -> Position
shiftPosition v (Cart x y z)
  = Cart (x + xComp v) (y + yComp v) (z + zComp v)

-- | An object is a map into 'Position'.
shiftObject :: Displacement -> (a -> Position) -> (a -> Position)
shiftObject d f = shiftPosition d . f

-- | A field is a map from 'Position'.
shiftField :: Displacement -> (Position -> v) -> (Position -> v)
shiftField d f = f . shiftPosition d
```

16.6 Scalar and vector fields

There are physical quantities, like volume charge density and electric potential, that are best described by giving a number for each point in space. These physical quantities are called scalar fields. The word *field* in physics means a function of physical space, something that can take a different value at each point in space. (The word *field* in mathematics means something else.) A scalar field is a field in which the value assigned at each point in space is a scalar, a number. Temperature is another example of a scalar field. The temperature in one place (Annville, Pennsylvania, for example) is usually different from the temperature at another place (Orlando, Florida, say).

Since a scalar field associates a number with each position in space, it makes sense to define a scalar field type to be a function from space to numbers. A vector field associates a vector with each point in space. Electric field and magnetic field are vector fields.

```
type ScalarField = Position -> R
type VectorField = Position -> Vec
```

Sometimes we want to be able to talk about a field without saying whether it is a scalar field or a vector field. Here we use the type variable `v` to stand for a type (probably the type will turn out to be `R` or `Vec`, but maybe there will be times when it could be either and we don't need to know).

```
type Field v = Position -> v
```

The unit vectors used with cylindrical and spherical coordinates, such as $\hat{\mathbf{s}}$, $\hat{\boldsymbol{\phi}}$, $\hat{\mathbf{r}}$, and $\hat{\boldsymbol{\theta}}$, are really *unit vector fields*, since their direction changes depending on their location in space.

The vector fields $\hat{\mathbf{s}}$ and $\hat{\boldsymbol{\phi}}$ are defined everywhere except on the z axis.

```
sHat   :: VectorField
sHat   r = vec ( cos phi) (sin phi) 0
    where
      (_,phi,_) = cylindricalCoordinates r

phiHat :: VectorField
phiHat r = vec (-sin phi) (cos phi) 0
    where
      (_,phi,_) = cylindricalCoordinates r
```

Here are definitions for the unit vector fields $\hat{\mathbf{r}}$ and $\hat{\boldsymbol{\theta}}$.

```
rHat :: VectorField
rHat rv = d ^/ magnitude d
    where
      d = displacement (cart 0 0 0) rv

thetaHat :: VectorField
thetaHat r = vec (cos theta * cos phi) (cos theta * sin phi) (-sin theta)
    where
      (_,theta,phi) = sphericalCoordinates r
```

We regard $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ as simple unit vectors (`Vec`s), but we define $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ as unit vector fields (`VectorField`s), in analogy with $\hat{\mathbf{s}}$, $\hat{\boldsymbol{\phi}}$, $\hat{\mathbf{r}}$, and $\hat{\boldsymbol{\theta}}$.

```
xHat :: VectorField
xHat = const iHat
```

```
yHat :: VectorField
yHat = const jHat

zHat :: VectorField
zHat = const kHat
```

Scalar and vector fields can be added. Here are some functions to do that.

```
-- | Add scalar fields.
addScalarFields :: [ScalarField] -> ScalarField
addScalarFields flds r = sum  [fld r | fld <- flds]

-- | Add vector fields.
addVectorFields :: [VectorField] -> VectorField
addVectorFields flds r = sumV [fld r | fld <- flds]
```

Chapter 17

Curves, Surfaces, and Volumes

Electrodynamics is a geometric subject. It will be useful for us to have types for curves, surfaces, and volumes.

Let's start coding. I like to turn warnings on so that the compiler will tell me when I'm doing something I may not have intended.

```
{-# OPTIONS_GHC -Wall #-}
```

Let's give the code in this chapter a module name, in case we want to import it later for use with some other code.

```
module Geometry where
```

We will use the type `Position` that we defined in the module `CoordinateSystems` in chapter 16, so we import that at the beginning of the Haskell code file.

```
import CoordinateSystems
    ( R
    , Position
    , cylindrical
    , spherical
    , cart
    , cyl
    , sph
    , shiftPosition
    , displacement
    )
import SimpleVec
    ( (*^)
    )
```

17.1 Curves

How can we describe a curve in space? We can parametrize the curve so there is a real number associated with each point on the curve, and then give (by way of a function) the position in space associated with each parametrized point on the curve. For example, a line along the y axis could be parametrized with the function

$$t \mapsto (0, t, 0).$$

A circle with radius 2 in the xy plane centered at the origin could be parametrized with the function

$$t \mapsto (2\cos t, 2\sin t, 0).$$

In these functions, t serves only as the name of a parameter (we could have chosen s or any convenient symbol) and has nothing to do with time.

A parametrized curve therefore requires a function with type `R -> Position` sending a parameter `t :: R` along the curve to a point `r :: Position` in space. But we also need starting and ending points for our curve. For example, the circle in the xy plane with radius 2 centered at the origin can be specified with the function

$$t \mapsto (2\cos t, 2\sin t, 0),$$

starting parameter $t_a = 0$, and ending parameter $t_b = 2\pi$.

If we use the same function and starting parameter, but change the ending parameter to $t_b = \pi$, we get a semicircle (the half circle above the x axis).

The starting and ending points can be specified by a starting parameter `startingCurveParam :: R` (which we called t_a above) and an ending parameter `endingCurveParam :: R` (which we called t_b above). We specify a curve, then, with three pieces of data: a function, a starting parameter, and an ending parameter.

Haskell's data types can be used to combine pieces of data that really belong together. For the curve, it will be very convenient to have a single type `Curve` that contains the function, the starting point, and the ending point.

```
data Curve = Curve { curveFunc          :: R -> Position
                   , startingCurveParam :: R
                   , endingCurveParam   :: R
                   }
```

The `Curve` that appears immediately to the right of the `data` keyword is the name we give to the new type. The `Curve` that appears to the right of the equals sign is called a *data constructor*. The type (`Curve` in this case) and the data constructor (`Curve` in this case) can have the same name or different names. They live in different *namespaces*, so that even if they have the same name (as they do in this case), they are not the same object. (Which is why we put the type in blue and the data constructor in black.)

Inside the curly braces, we see the three pieces of data, along with their types, that make up the curve type. Let's play with this in GHCi.

GHCi
```
:l Charge.lhs
```

GHCi
```
:t curveFunc
   ⇝  curveFunc :: Curve -> R -> Position
```

Let's think about the type of `curveFunc`. It takes a `Curve` and produces a function `R -> Position`. It is giving us the `curveFunc` of a particular `Curve`. Let's code the example of the circle with radius 2 in the xy plane centered at the origin.

```
circle2 :: Curve
circle2 = Curve (\t -> cart (2 * cos t) (2 * sin t) 0) 0 (2*pi)
```

We are naming our curve `circle2` (to remind us of the radius 2). The first line above is a declaration, declaring that `circle2` has the type (which we just created) `Curve`. The second line is the definition of `circle2`. To the right of the equals sign we use the data constructor `Curve` followed by the three data pieces. Here we have chosen to give the three pieces in order, without reference to their names. There is also an alternative syntax you can use to give the data pieces by name. Note the use of an anonynous function to specify the curve function. We need parentheses around the anonymous function and the `2*pi` so that the compiler can clearly identify the three data pieces.

A circle in the xy plane centered at the origin is easier to express in cylindrical coordinates than in Cartesian. In cylindrical coordinates, our circle has the constant values $s = 2$ and $z = 0$. Only the ϕ coordinate changes from 0 to 2π. This suggests that we use the ϕ coordinate as our parameter for the curve.

```
circle2' :: Curve
circle2' = Curve (\phi -> cyl 2 phi 0) 0 (2*pi)
```

Note the `cyl` function used to specify the curve in cylindrical coordinates. The curve `circle2'` is the same as the curve `circle2`. Unfortunately, Haskell cannot check that this is true, because it cannot compare functions for equality. It is true nonetheless.

17.2 Surfaces

To describe surface charge density, we will first need a way to describe a surface in space. A surface is a parametrized function from two parameters to space. For example, we can parametrize the unit sphere with two parameters, θ and ϕ as the function

$$(\theta, \phi) \mapsto (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)$$

and the ranges $0 \le \theta \le \pi$ and $0 \le \phi \le 2\pi$.

For a second example, suppose we want to parametrize the surface that lies in the xy plane, bounded by the parabola $y = x^2$ and the line $y = 4$. This surface is shown in Figure 17.1. In this case, it makes sense to use x and y as the parameters. The parametrized function for the surface is not very exciting.

$$(x, y) \mapsto (x, y, 0)$$

The interesting part about this surface is the specification of the boundary. There is a lower curve $y = x^2$ that gives the bottom boundary, an upper

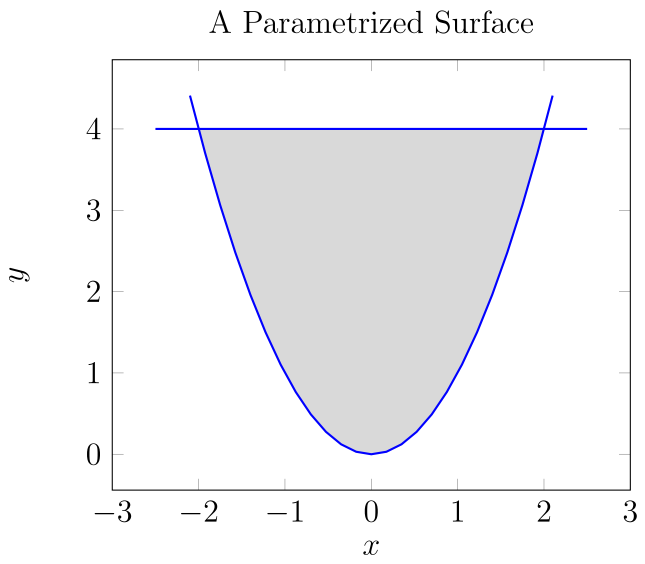


Figure 17.1: A parametrized surface

curve $y = 4$ that gives that gives the top boundary, a lower limit of $x = -2$ that specifies the left boundary, and an upper limit of $x = 2$ that specifies the right boundary.

For a general surface, we will call our two parameters s and t. To specify a general surface, there are five pieces of data we must give: a parametrizing function of two variables, a lower curve, an upper curve, a lower limit, and an upper limit. Here is the data type definition for a general surface.

```
data Surface = Surface { surfaceFunc :: (R,R) -> Position
                       , lowerLimit  :: R        -- ^ s_l
                       , upperLimit  :: R        -- ^ s_u
                       , lowerCurve  :: R -> R   -- ^ t_l(s)
                       , upperCurve  :: R -> R   -- ^ t_u(s)
                       }
```

The function `surfaceFunc` is the parametrizing function that maps (s, t) into a `Position`. The lower curve is given as a function $t_l(s)$ that gives the lowest value of t on the surface for each value of the parameter s. The upper curve is given as a function $t_u(s)$ that gives the highest value of t on the surface for each value of the parameter s. The lower limit s_l is the lowest value of s on the surface, and the upper limit s_u is the largest value of s on the surface.

To code the unit sphere we discussed above, we can do the following.

```haskell
unitSphere :: Surface
unitSphere = Surface (\(th,phi) -> cart (sin th * cos phi)
                                        (sin th * sin phi)
                                        (cos th))
                     0 pi (const 0) (const $ 2*pi)
```

In this case, we want constant functions for the lower and upper curves, so we use the `const` function to turn a number into a constant function.

Not surprisingly, it's easier to specify a unit sphere in spherical coordinates.

```haskell
unitSphere' :: Surface
unitSphere' = Surface (\(th,phi) -> sph 1 th phi)
                      0 pi (const 0) (const $ 2*pi)
```

In spherical coordinates, we used the same parameters (θ, ϕ), the same lower and upper curves, and the same limits. Only the parametrizing function changes. The surface `unitSphere'` is the same surface as `unitSphere`. Again, Haskell cannot confirm this for us, but it is true.

Let's code up our parabola surface from Figure 17.1.

```haskell
parabolaSurface :: Surface
parabolaSurface = Surface (\(x,y) -> cart x y 0)
                          (-2) 2 (\x -> x*x) (const 4)
```

Here are some other examples of surfaces. The first is a sphere with given radius centered at the origin. The second is the northern hemisphere of the unit sphere. The third is a disk in the xy plane.

```haskell
centeredSphere :: R -> Surface
centeredSphere r = Surface (\(th,phi) -> sph r th phi)
```

```
                              0 pi (const 0) (const $ 2*pi)

northernHemisphere :: Surface
northernHemisphere = Surface (\(th,phi) -> sph 1 th phi)
                               0 (pi/2) (const 0) (const $ 2*pi)

disk :: R -> Surface
disk radius = Surface (\(s,phi) -> cyl s phi 0)
                        0 radius (const 0) (const (2*pi))
```

17.3 Volumes

When we have charge that is distributed throughout a volume, we will use a volume charge density to describe this, and we will need a new data type to describe a volume. We need to specify seven pieces of data to describe a volume. These are (i) a parametrizing function from three parameters (s, t, u) into space, (ii) a lower surface $u_l(s, t)$ describing the lowest value of u for each (s, t), (iii) an upper surface $u_u(s, t)$ describing the highest value of u for each (s, t), (iv) a lower curve $t_l(s)$ describing the lowest value of t for each value of s, (v) an upper curve $t_u(s)$ describing the highest value of t for each value of s, (vi) a lower limit s_l describing the lowest value of s, and (vii) an upper limit s_u describing the highest value of s.

Here is the definition of the `Volume` data type.

```
data Volume = Volume { volumeFunc :: (R,R,R) -> Position
                     , loLimit    :: R              -- ^ s_l
                     , upLimit    :: R              -- ^ s_u
                     , loCurve    :: R -> R         -- ^ t_l(s)
                     , upCurve    :: R -> R         -- ^ t_u(s)
                     , loSurf     :: R -> R -> R    -- ^ u_l(s,t)
                     , upSurf     :: R -> R -> R    -- ^ u_u(s,t)
                     }
```

The `volumeFunc` has type `(R,R,R) -> Position`. Recall from Chapter 16 that this type is the same as `CoordinateSystem`. We will often want to use `cartesian`, `cylindrical`, or `spherical` as our `volumeFunc`, although it is possible to invent your own coordinate system.

17.3.1 A unit ball

As a first example, here is a unit ball, centered at the origin.

```
unitBall :: Volume
unitBall = Volume spherical 0 1 (const 0) (const pi)
                  (\_ _ -> 0) (\_ _ -> 2*pi)
```

For the `volumeFunc`, we use `spherical`, which means that the parameters (s, t, u) are the spherical coordinates (r, θ, ϕ). We must provide a lower limit r_l, an upper limit r_u, a lower curve $\theta_l(r)$, an upper curve $\theta_u(r)$, a lower surface $\phi_l(r, \theta)$, and an upper surface $\phi_u(r, \theta)$. For a ball, we should pick

$$\begin{aligned} r_l &= 0 \\ r_u &= 1 \\ \theta_l(r) &= 0 \\ \theta_u(r) &= \pi \\ \phi_l(r,\theta) &= 0 \\ \phi_u(r,\theta) &= 2\pi \end{aligned}$$

Notice that θ_l is the function $r \mapsto 0$ (in Haskell notation `\r -> 0` or `\_ -> 0`). This the same as the constant function that returns 0 for any input (in Haskell notation `const 0`). The function ϕ_l takes *two* inputs and returns 0 (in Haskell notation `\_ _ -> 0`).

Exercise 17.1. Replace the `undefined` below with a definition of an upper half ball ($z \geq 0$) with unit radius, centered at the origin.

```
northernHalfBall :: Volume
northernHalfBall = undefined
```

Exercise 17.2. Replace the `undefined` below with a definition of a ball with given radius, centered at the origin. (The `R` is the type of the radius, and you may want to put a variable for the radius on the left of the equals sign.)

```
centeredBall :: R -> Volume
centeredBall = undefined

-- | Cylinder with given radius and height.  Circular base of the cylinder
--   is centered at the origin.  Circular top of the cylinder lies in plane z = h.
centeredCylinder :: R        -- radius
                 -> R        -- height
                 -> Volume   -- cylinder
centeredCylinder r h = Volume cylindrical 0 r (const 0) (const (2*pi)) (\_ _ -> 0) (\

-- | The straight-line curve from one position to another.
straightLine :: Position  -- ^ starting position
             -> Position  -- ^ ending position
             -> Curve     -- ^ straight-line curve
straightLine r1 r2 = Curve f 0 1
    where
      f t = shiftPosition (t *^ d) r1
      d = displacement r1 r2
```

Chapter 18

Electric Charge

Some particles have electric charge. We say that protons are positively charged, electrons are negatively charged, and neutrons have no net charge. Photons and neutrinos have no charge. All of the particles that we know of that have charge also have mass.

18.1 Charge Distributions

Electric charge is the fundamental quantity responsible for electromagnetic effects and plays a key role in electromagnetic theory. In classical electromagnetic theory, which we study in this book, we will sometimes think of charge as associated with a particle, in which case we call the charge a *point charge*, imagining that it has a location in space, but no spatial extent. The SI unit for charge is the Coulomb (C). The Coulomb is a very large unit of charge. The charge of a proton, for example, is 1.602×10^{-19} C, a very small number of Coulombs. We typically use the symbols q and Q for charge.

In classical electromagnetic theory, we also want to sometimes think of charge as a fluid, something that can be continuously distributed throughout a region of space. In fact, there are three types of continuous charge distributions that we use. First, there is charge continously distributed along a one-dimensional path such as a line or a curve. In this case, we speak of the *linear charge density* λ (greek letter lambda), meaning the charge per unit length. The SI unit for linear charge density is the Coulomb per meter (C/m).

Second, there is charge continously distributed along a two-dimensional

Charge distribution	Dimensionality	Symbol	SI unit
Point charge	0	q, Q	C
Linear charge density	1	λ	C/m
Surface charge density	2	σ	C/m^2
Volume charge density	3	ρ	C/m^3

Table 18.1: Charge distributions

surface. In this case, we speak of the *surface charge density* σ (greek letter sigma), meaning the charge per unit area. The SI unit for surface charge density is the Coulomb per square meter (C/m^2). Finally, there is charge continously distributed throughout a three-dimensional volume. In this case, we speak of the *volume charge density* ρ (greek letter rho), meaning the charge per unit volume. The SI unit for volume charge density is the Coulomb per cubic meter (C/m^3). These charge distributions are summarized in Table 18.1.

By the end of this chapter, we will have in hand a new type capable of holding point charges, charge densities of any dimensionality, and combinations of distributions (like a point charge and a surface charge). We will call this new type `ChargeDistribution`. Since charge is the fundamental source of electromagnetic effects, this `ChargeDistribution` type will play an important role. We will write a function to find the total charge of a `ChargeDistribution`, and in Chapter 19 we will write a function to calculate the electric field produced by a `ChargeDistribution`. Let's start coding.

I like to turn warnings on so that the compiler will tell me when I'm doing something I may not have intended.

```
{-# OPTIONS_GHC -Wall #-}
```

Let's give the code in this chapter a module name, in case we want to import it later for use with some other code.

```
module Charge where
```

We will use types and functions defined in Chapters 16 and 17 as well as Appendix C, so we import those types and functions at the beginning of the Haskell code file.

```
import CoordinateSystems
    ( R
    , Position
    , ScalarField
    , cart
    )
import Geometry
    ( Curve(..)
    , Surface(..)
    , Volume(..)
    )
import VectorIntegrals
    ( scalarLineIntegral
    , scalarSurfaceIntegral
    , scalarVolumeIntegral
    )
```

Now that we are talking about charge, I am going to define a type synonym for charge.

```
type Charge = R
```

Defining a new type for charge in this way is half good and half silly. It's good in that human readers of the code (including the writer of the code) will know the intent of an expression with type `Charge`. It is, in this sense, a form of documentation for the code. It is silly because the compiler doesn't make any distinction between `Charge` and `R` and `Double`, so it cannot help the writer to avoid using charge in any place in which an `R` or a `Double` could be used. One of the main purposes of types is separating things that should be separated, and letting the computer help enforce that separation. Charge is not at all the same as time, for example, which would also probably be described by a real number `R`. Haskell has a mechanism to define a new type that will not be confused with any other type. (The way to do this uses Haskell's `data` keyword instead of the `type` keyword. Haskell allows the construction of sophisticated algebraic data types that provide the power to separate ideas that should be separated.) Defining a new data type would be a very reasonable thing to do here, but there is a bit of extra effort and overhead involved, and so I have chosen the simplicity of the `type` method over the power of the `data` method.

Defining `R` to be the same as `Double` (which we did in chapter 16 in the module `CoordinateSystems`) is completely reasonable. We're just saying that there is another name we'd rather use. Defining `Charge` to be the same as `R` is only half reasonable. Charge is only one of several physical quantities that we want to use real numbers to represent.

What about the continuous charge densities? What information is required to specify these?

Let's start with linear charge density. To describe continuous charge that is (not necessarily uniformly) spread over a one-dimensional curve, we need to (a) describe the geometry of the curve in space, and (b) give a value for the linear charge density (in C/m, say) at each point on the curve.

For item (a), we will use the `Curve` type that we defined in Chapter 17. For item (b), to describe a linear charge density, we need to give its value at each point on the curve. There are two ways to do this. Option 1 is to give the linear charge density value as a function of the curve parameter. Option 2 is to give the value as a function of spatial coordinates. We will choose option 2. This means that to specify a linear charge density, we must give a `ScalarField` along with a `Curve`.

To specify a surface charge density, we must give a `ScalarField` along with a `Surface`. For a volume charge density, we need a `ScalarField` along with a `Volume`.

18.2 A type for charge distribution

A charge distribution is a point charge, a line charge, a surface charge, a volume charge, or a combination of these. The `ScalarField` describes a linear charge density, a surface charge density, or a volume charge density. For a point charge, we need to specify the position. Location information for the continous densities is included in the `Curve`, `Surface`, or `Volume`. We also give a way to combine multiple charges (of arbitrary types) into a single charge distribution.

```
data ChargeDistribution
    = PointCharge   Charge      Position
    | LineCharge    ScalarField Curve
    | SurfaceCharge ScalarField Surface
    | VolumeCharge  ScalarField Volume
    | MultipleCharges [ChargeDistribution]
```

Let's write some examples of charge distribitions. The charge distribution of a proton at the origin can be defined as follows.

```
protonOrigin :: ChargeDistribution
protonOrigin = PointCharge 1.602e-19 (cart 0 0 0)
```

Here I am using SI units (giving the charge in Coulombs).

A line segment of charge lying on the x axis from $x = -1$ m to $x = 1$ m with a uniform linear charge density $\lambda = 0.5$ C/m can be written as follows.

```
positiveStick :: ChargeDistribution
positiveStick = LineCharge (const 0.5)
                           (Curve (\t -> cart t 0 0) (-1) 1)
```

18.3 Total charge

Here is a function to calculate the total charge of any charge distribution. For line charges, we use a scalar line integral. For surface charges, we use a scalar surface integral. For volume charges, we use a scalar volume integral. These integrals are defined and described in Appendix C. For a combination of multiple charges (which could be of different sorts, like a point charge and a surface charge) we simply add the total charges of each part.

```
totalCharge :: ChargeDistribution -> Charge
totalCharge (PointCharge   q      _)
    = q
totalCharge (LineCharge    lambda c)
    = scalarLineIntegral   1000    lambda c
totalCharge (SurfaceCharge sigma  s)
    = scalarSurfaceIntegral 200 200 sigma s
```

```
totalCharge (VolumeCharge  rho    v)
    = scalarVolumeIntegral  50  50 50 rho v
totalCharge (MultipleCharges ds    )
    = sum [totalCharge d | d <- ds]
```

Let's check the total charge of the distributions we defined above.

```
GHCi  :l Charge.lhs
GHCi  totalCharge protonOrigin
          ⇝  1.602e-19
GHCi  totalCharge positiveStick
          ⇝  1.0000000000000009
```

Chapter 19

Electric Field

In the 1700s, people discovered that there were two types of electric charge. Charges of the same type repelled each other and charges of different types attracted each other. It was convenient to call one type of charge positive and the other type negative. The proton ended up positive and the electron negative, but that was an arbitrary choice that everyone now respects as a convention.

19.1 Coulomb's law

Coulomb was the first to give a quantitative relationship describing the interaction of two charged particles. He showed that the force exerted by one point charge on another is proportional to each charge, and inversely proportional to the square of the distance between them. As an equation, Coulomb's law can be written

$$F = k\frac{|q_1 q_2|}{r^2} \tag{19.1}$$

where q_1 is the charge of particle 1, q_2 is the charge of particle 2, and r is the distance between the particles. This equation gives the magnitude of the force produced by particle 2 on particle 1 (which, by Newton's third law, is the same as the magnitude of the the force produced by particle 1 on particle 2). The direction of the force depends on the signs of the charges; the force is repulsive for like charges and attractive for unlike charges. In SI units, the constant $k = 9 \times 10^9$ N m^2/C^2 (approximately).

We can use vector notation to give a more comprehensive version of Coulomb's law, which includes the direction of the force in the equation.

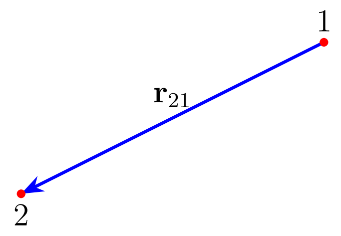


Figure 19.1: Definition of displacement vector $\mathbf{r}_{21}$

Define the displacement vector $\mathbf{r}_{21}$ to be the vector that points from particle 1 to particle 2. (See Figure 19.1.)

The force $\mathbf{F}_{21}$ exerted *on* particle 2 produced *by* particle 1 is given in vector notation as follows.

$$\mathbf{F}_{21} = kq_1q_2\frac{\mathbf{r}_{21}}{|\mathbf{r}_{21}|^3} \tag{19.2}$$

Notice that if both charges are positive, then the force $\mathbf{F}_{21}$ on particle 2 points in the same direction as the displacement vector $\mathbf{r}_{21}$ (away from particle 1, as we expect for like charges). If the charges have unlike signs, then the direction of $\mathbf{F}_{21}$ will flip, indicating an attractive force. In summary, Coulomb's law 19.1 is simpler, and Coulomb's law 19.2 is more powerful, since the direction of the force is encoded in the equation.

19.2 Electric Field

In the 1800s, Faraday and Maxwell discovered a new way to think about electric (and magnetic) phenomena, a way that forms the basis for today's electromagnetic theory. In this 19th century view, one particle does not directly apply a force to another particle, as Coulomb's law would imply. Instead, one particle creates an electric field, and the electric field applies a force to the second particle.

What is this electric field? The electric field is a *vector field* of the kind we talked about in Section 16.6. The electric field is composed of a vector at each point in space that describes (almost) what the force would be on a particle if there were a particle at that point in space.

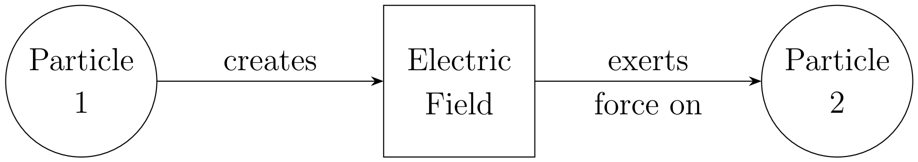


Figure 19.2: Conceptual diagram of the role of the electric field when two charged particles are present

Let us explain the Coulomb situation in Figure 19.1 in terms of electric field. Particle 1 creates an electric field **E**. This electric field specifies a vector for each point **r** in space, given by the following equation.

$$\mathbf{E}(\mathbf{r}) = k q_1 \frac{\mathbf{r} - \mathbf{r}_1}{|\mathbf{r} - \mathbf{r}_1|^3} \tag{19.3}$$

The force on a particle with charge q sitting in an electric field **E** is

$$\mathbf{F} = q\mathbf{E} \tag{19.4}$$

Where by **E** we mean the particular electric field vector at the location of the particle.

$$\mathbf{F}_2 = q_2 \mathbf{E}(\mathbf{r}_2) \tag{19.5}$$

$$\mathbf{F}_{21} = q_2 \mathbf{E}(\mathbf{r}_2) = k q_1 q_2 \frac{\mathbf{r}_2 - \mathbf{r}_1}{|\mathbf{r}_2 - \mathbf{r}_1|^3} = k q_1 q_2 \frac{\mathbf{r}_{21}}{|\mathbf{r}_{21}|^3} \tag{19.6}$$

and we get back the Coulomb result. Why introduce the electric field if we just get back Coulomb's result?

Talk about symmetry. Particle 2 also contributes to E.

```
{-# OPTIONS_GHC -Wall #-}

module ElectricField where

import CoordinateSystems
    ( Position
    , ScalarField
```

```haskell
    , VectorField
    , displacement
    , addScalarFields
    , addVectorFields
    , cart
    )
import Geometry
    ( Curve(..)
    , Surface(..)
    , Volume(..)
    , straightLine
    )
import Charge
    ( Charge
    , ChargeDistribution(..)
    , positiveStick
    )
import SimpleVec
    ( Vec
    , (*^)
    , (^/)
    , magnitude
    )
import VectorIntegrals
    ( scalarLineIntegral
    , vectorLineIntegral
    , dottedLineIntegral
    , scalarSurfaceIntegral
    , vectorSurfaceIntegral
    , dottedSurfaceIntegral
    , scalarVolumeIntegral
    , vectorVolumeIntegral
    )

-- | Electric field produced by a point charge.
--   The function 'eField' calls this function
--   to evaluate the electric field produced by a point charge.
```

```
eFieldFromPointCharge
    :: Charge             -- ^ charge (in Coulombs)
    -> Position           -- ^ of point charge
    -> VectorField        -- ^ electric field (in V/m)
eFieldFromPointCharge q r' r
    = (k * q) *^ d ^/ magnitude d ** 3
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        d = displacement r' r

-- | Electric field produced by a line charge.
--   The function 'eField' calls this function
--   to evaluate the electric field produced by a line charge.
eFieldFromLineCharge
    :: ScalarField        -- ^ linear charge density lambda
    -> Curve              -- ^ geometry of the line charge
    -> VectorField        -- ^ electric field (in V/m)
eFieldFromLineCharge lambda c r
    = k *^ vectorLineIntegral 1000 integrand c
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = lambda r' *^ d ^/ magnitude d ** 3
            where
              d = displacement r' r

-- | Electric field produced by a surface charge.
--   The function 'eField' calls this function
--   to evaluate the electric field produced by a surface charge.
eFieldFromSurfaceCharge
    :: ScalarField        -- ^ surface charge density sigma
    -> Surface            -- ^ geometry of the surface charge
    -> VectorField        -- ^ electric field (in V/m)
eFieldFromSurfaceCharge sigma s r
    = k *^ vectorSurfaceIntegral 200 200 integrand s
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = sigma r' *^ d ^/ magnitude d ** 3
```

```
            where
              d = displacement r' r

-- | Electric field produced by a volume charge.
--   The function 'eField' calls this function
--   to evaluate the electric field produced by a volume charge.
eFieldFromVolumeCharge
    :: ScalarField     -- ^ volume charge density rho
    -> Volume          -- ^ geometry of the volume charge
    -> VectorField     -- ^ electric field (in V/m)
eFieldFromVolumeCharge rho v r
    = k *^ vectorVolumeIntegral 50 50 50 integrand v
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = rho r' *^ d ^/ magnitude d ** 3
            where
              d = displacement r' r

-- | The electric field produced by a charge distribution.
--   This is the simplest way to find the electric field, because it
--   works for any charge distribution (point, line, surface, volume, or combi
eField :: ChargeDistribution -> VectorField
eField (PointCharge q r') = eFieldFromPointCharge q r'
eField (LineCharge lam c) = eFieldFromLineCharge lam c
eField (SurfaceCharge sig s) = eFieldFromSurfaceCharge sig s
eField (VolumeCharge rho v) = eFieldFromVolumeCharge rho v
eField (MultipleCharges cds) = addVectorFields $ map eField cds

-------------------
-- Electric Flux --
-------------------

-- | The electric flux through a surface produced by a charge distribution.
electricFlux :: Surface -> ChargeDistribution -> Double
electricFlux surf dist = dottedSurfaceIntegral 200 200 (eField dist) surf

------------------------
```

```
-- Electric Potential --
------------------------

-- | Electric potential from electric field, given a position to be the zero
--   of electric potential.
electricPotentialFromField :: Position     -- ^ position where electric potential is 
                           -> VectorField  -- ^ electric field
                           -> ScalarField  -- ^ electric potential
electricPotentialFromField base ef r = -dottedLineIntegral 1000 ef (straightLine base

-- | Electric potential produced by a charge distribution.
--   The position where the electric potential is zero is taken to be infinity.
electricPotentialFromCharge :: ChargeDistribution -> ScalarField
electricPotentialFromCharge (PointCharge q r') = ePotFromPointCharge q r'
electricPotentialFromCharge (LineCharge lam c) = ePotFromLineCharge lam c
electricPotentialFromCharge (SurfaceCharge sig s) = ePotFromSurfaceCharge sig s
electricPotentialFromCharge (VolumeCharge rho v) = ePotFromVolumeCharge rho v
electricPotentialFromCharge (MultipleCharges cds) = addScalarFields $ map electricPot

ePotFromPointCharge
    :: Charge           -- ^ charge (in Coulombs)
    -> Position         -- ^ of point charge
    -> ScalarField      -- ^ electric potential
ePotFromPointCharge q r' r
    = (k * q) / magnitude d
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        d = displacement r' r

ePotFromLineCharge
    :: ScalarField      -- ^ linear charge density lambda
    -> Curve            -- ^ geometry of the line charge
    -> ScalarField      -- ^ electric potential
ePotFromLineCharge lambda c r
    = k * scalarLineIntegral 1000 integrand c
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = lambda r' / magnitude d
            where
              d = displacement r' r

ePotFromSurfaceCharge
    :: ScalarField     -- ^ surface charge density sigma
    -> Surface         -- ^ geometry of the surface charge
    -> ScalarField     -- ^ electric potential
ePotFromSurfaceCharge sigma s r
    = k * scalarSurfaceIntegral 200 200 integrand s
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = sigma r' / magnitude d
            where
              d = displacement r' r

ePotFromVolumeCharge
    :: ScalarField     -- ^ volume charge density rho
    -> Volume          -- ^ geometry of the volume charge
    -> ScalarField     -- ^ electric potential
ePotFromVolumeCharge rho v r
    = k * scalarVolumeIntegral 50 50 50 integrand v
      where
        k = 9e9  -- 1 / (4 * pi * epsilon0)
        integrand r' = rho r' / magnitude d
            where
              d = displacement r' r

eTest :: Vec
eTest = eField positiveStick (cart 0 0 1)
```

19.3 Electric Field produced by a Line Segment of Charge

Imagine a line segment of charge, with linear charge density λ. Let us place this line segment on the x axis from $x = -L/2$ to $x = L/2$. We wish to find the electric field produced by this line segment at some point $\mathbf{r} = x\hat{\mathbf{i}} + y\hat{\mathbf{j}}$.

The electric field at point $\mathbf{r} = x\hat{\mathbf{i}} + y\hat{\mathbf{j}}$ is given by

$$\mathbf{E}(x,y) = \int_{-L/2}^{L/2} \frac{1}{4\pi\epsilon_0} \frac{\lambda dx'}{[(x-x')^2 + y^2]^{3/2}} \left[(x-x')\hat{\mathbf{i}} + y\hat{\mathbf{j}} \right].$$

You should be able to derive this expression. If it is not clear how this expression comes about, you should look in an introductory physics textbook or ask me.

To find the electric field, we will do numerical integration. We can use the numerical integrator that we wrote in the last section.

Activity 19.1. Write a program to show the electric field produced by a line segment of charge. You may restrict your attention to the xy plane. Plot the electric field vectors that you calculate with numerical integration in red.

The electric field produced by a line segment of charge is an exactly solvable problem in physics. Find the exact solution in an introductory physics textbook. (Or do the integral above yourself, if you like.) Plot the electric field vectors obtained from the exact solution in blue. If you have a good numerical integrator, you should get good agreement between the numerical (red) and exact (blue) electric field vectors.

To get started, you can download a module to make arrows, and a template for the electric field project.

```
wget http://quantum.lvc.edu/walck/phy261/Arrow.hs
wget http://quantum.lvc.edu/walck/phy261/eFieldFromLineTemplate.hs
```

Note that the template is set up to show only one set of arrows (for example, the exactly calculated arrows, but not the numerically integrated arrows). You will need to extend it to display both sets of arrows.

Chapter 20

Electric Current

Electric current is the flow or movement of electric charge.

20.1 Current Distributions

Electric current is the fundamental quantity responsible for magnetic effects (although this took thousands of years to discover after magnetic phenomena were first observed) and plays a key role in electromagnetic theory. The most common way to think about current is charge flowing along a wire (a one-dimensional path).

But current can also flow along a surface (two dimensional), or throughout a volume (three-dimensional). Since charge must move through space in order to have a current, we cannot have a point current (zero dimensional).

There are three types of current distributions that we use. First, there is current flowing along a one-dimensional path such as a line or a curve.

The SI unit for current is the Ampere or Amp (A). An ampere of current in a wire means that one Coulomb of charge is passing a fixed point on the wire in each second. We typically use the symbol I for current.

Second, there is current flowing along a two-dimensional surface. In this case, we speak of the *surface current density* $\mathbf{K}$, meaning the current per unit of cross-section length. The SI unit for surface current density is the Ampere per meter (A/m). Finally, there is current flowing throughout a three-dimensional volume. In this case, we speak of the *volume current density* $\mathbf{J}$, meaning the current per unit of cross-sectional area. The SI unit for volume current density is the Ampere per square meter (A/m^2). These

Charge distribution	Dimensionality	Symbol	SI unit
Point current	0	not possible	
Current	1	I	A
Surface current density	2	$\mathbf{K}$	A/m
Volume current density	3	$\mathbf{J}$	A/m^2

Table 20.1: Current distributions

current distributions are summarized in Table 20.1.

Let's start coding. I like to turn warnings on so that the compiler will tell me when I'm doing something I may not have intended.

```
{-# OPTIONS_GHC -Wall #-}
```

Let's give the code in this chapter a module name, in case we want to import it later for use with some other code.

```
module Current where
```

We will use the type `Position` that we defined in the module `CoordinateSystems` in chapter 16, so we import that at the beginning of the Haskell code file.

```
import CoordinateSystems


import Physics.Learn.CarrotVec
    ( magnitude
    , (*^)
    , (^/)
    , (><)
    )
import Physics.Learn.Position
    ( VectorField
    , displacement
    , addFields
    )
import Physics.Learn.Curve
```

```
    ( Curve(..)
    , crossedLineIntegral
    )
import Physics.Learn.Surface
    ( Surface(..)
    , surfaceIntegral
    , dottedSurfaceIntegral
    )
import Physics.Learn.Volume
    ( Volume(..)
    , volumeIntegral
    )
```

Now that we are talking about current, I am going to define a type synonym for current.

```
type Current = R
```

This is entirely analogous to the type synonym we made for `Charge`. It is simple, because we want current to be just a number, but because `Current`, `Charge`, and `R` are really all the same under the hood, the compiler will not be able to help us from mistakenly using a `Charge` where a `Current` should go or vice versa.

20.2 A type for current distribution

```
-- | A current distribution is a line current
--   (current through a wire), a surface current,
--   a volume current, or a combination of these.
--   The 'VectorField' describes a surface current
--   density or a volume current density.
data CurrentDistribution
  = LineCurrent    Current     Curve
  | SurfaceCurrent VectorField Surface
  | VolumeCurrent  VectorField Volume
  | MultipleCurrents [CurrentDistribution]
```

20.3 Total current

$$I = \int \mathbf{J} \cdot d\mathbf{a}$$

Chapter 21

Magnetic Field

A simple magnetic effect is that two parallel wires carrying current in the same direction will attract each other. In fact, the SI system of units *defines* the Ampere to be that current which, when maintained in two parallel wires of infinite length, one meter apart, will produce an attractive force of 2×10^{-7} N per meter of wire length. The unit of current is defined in terms of its magnetic effect.

In the 1800s, Faraday and Maxwell discovered a new way to think about electric (and magnetic) phenomena, a way that forms the basis for today's electromagnetic theory. In this 19th century view, one current does not directly apply a force to another current. Instead, one current creates a magnetic field, and the magnetic field applies a force to the second current, as in Figure 21.1.

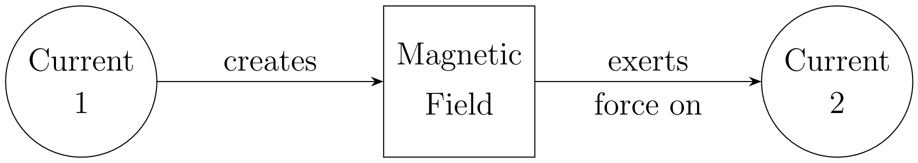


Figure 21.1: Conceptual diagram of the role of the magnetic field when two currents are present

21.1 Magnetic Field of a Circular Current Loop

One of the simplest and most natural ways to produce a magnetic field is with a circular loop of current. A circular loop of current is also a nice model of a magnetic dipole, which is a fundamental source of magnetic field. Surprisingly, there is no analytical solution for the magnetic field produced by a circular current loop. However, we can get a good approximate solution using numerical integration.

Consider a circular loop in the xz plane, centered at the origin, with radius R. This loop carries a current I in a counter-clockwise direction when viewed from the positive y axis.

Question 21.1. Use the Biot-Savart law to come up with an integral that gives the magnetic field at a point $\mathbf{r} = x\hat{\mathbf{i}} + y\hat{\mathbf{j}}$. Feel free to check this result with me before proceeding.

Activity 21.1. Write a program to show the magnetic field produced by a circular current loop. You may restrict your attention to calculating the magnetic field at points in the xy plane.

You will need to make some design decisions, such as how to scale the magnetic field vectors and at which points to plot the magnetic field.

Chapter 22

Motion of a Charged Particle

Maxwell's 1865 insight into the theory of electricity and magnetism was that electric and magnetic fields were actors on the stage of physics with the same stature as particles.

One of the important ways that

Electric and magnetic fields produce forces on a charged particle. The force on a particle with charge q and velocity $\mathbf{v}$ in an electric field $\mathbf{E}$ and a magnetic field $\mathbf{B}$ is given by the Lorentz force law.

$$\mathbf{F} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B})$$

External means produced by *other* charges and currents.

In some ways, this is the easier side of electromagnetic theory.

```
{-# OPTIONS_GHC -Wall #-}

module Main where

import Physics.Learn
import Vis
import SpatialMath
    ( Euler(..)
    )

drawFunction :: SimpleState -> VisObject Double
drawFunction (_t,r,_v)
```

```
    = RotEulerDeg (Euler 270 0 0) $ RotEulerDeg (Euler 0 180 0) $
      VisObjects [ Axes (0.5, 15)
                 , Trans (v3FromPos r) (Sphere 0.1 Solid red)
                 ]

statePropagationFunction :: Float -> SimpleState -> SimpleState
statePropagationFunction t' (t,r,v) = rungeKutta4 newton2 (realToFrac t' - t)

-- Newton's Second Law
newton2 :: SimpleState -> Diff SimpleState
newton2 (t,r,v) = (1,v,force (t,r,v) ^/ m)

-- Lorentz Force Law
force :: SimpleState -> Vec
force (_t,r,v) = q *^ (electricField r ^+^ v >< magneticField r)

myOptions :: Options
myOptions = defaultOpts {optWindowName = "Particle Experiencing Electromagneti

main :: IO ()
main = simulate
       myOptions
       0.01
       (0,initialPosition,initialVelocity)
       drawFunction
       statePropagationFunction

-- particle mass
m :: Double
m = 1

-- particle charge
q :: Double
q = 1

-- Electric Field
electricField :: VectorField
electricField r = vec 0 2 0
    where
      (x,y,z) = cartesianCoordinates r

-- Magnetic Field
magneticField :: VectorField
magneticField r = vec 0 0 4
    where
      (x,y,z) = cartesianCoordinates r

-- Initial displacement
initialPosition :: Position
initialPosition = cart 0 0 0

-- Initial velocity
initialVelocity :: Vec
initialVelocity = vec 0 0 0
```

Appendix A

Color Summary

Entity	Color	Example
number	black	`4.2`
character	black	`'H'`
string	grey	`"Haskell"`
constant name	black	`e`
function name	black	`square`
anonymous function	black	`\x -> x*x`
data constructor	black	`True`
comment	orange	`-- a comment`
long comment	orange	`{- a long comment -}`
type	blue	`Double -> Double`
type variable	blue	`a`
type constructor	blue	`Maybe`
kind	red	`* -> *`
keyword	bright purple	`if`
type class	green	`Num`
module name	light brown	`Graphics.Gnuplot.Simple`
pragma	pink	`{-# OPTIONS_GHC -Wall #-}`

Appendix B

A Type for Vectors

This module defines a type `Vec` for three-dimensional vectors, along with associated vector functions such as vector addition and scalar multiplication. This module is simple in the sense that the operations on vectors all have simple, concrete types, without the need for type classes. This makes using and reasoning about vector operations easier for a person just learning Haskell.

```
{-# OPTIONS_GHC -Wall #-}

module SimpleVec
    ( Vec(..)
    , R
    , vec
    , (^+^)
    , (^-^)
    , (*^)
    , (^*)
    , (^/)
    , (<.>)
    , (><)
    , magnitude
    , zeroV
    , negateV
    , sumV
    , iHat
    , jHat
    , kHat
    )
    where

infixl 6 ^+^
infixl 6 ^-^
infixl 7 *^
infixl 7 ^*
infixl 7 ^/
infixl 7 <.>
infixl 7 ><

type R = Double

-- | A type for vectors.
data Vec = Vec { xComp :: R  -- ^ x component
               , yComp :: R  -- ^ y component
               , zComp :: R  -- ^ z component
               } deriving (Eq)

instance Show Vec where
    show (Vec x y z) = "vec " ++ showDouble x ++ " "
                              ++ showDouble y ++ " "
                              ++ showDouble z

showDouble :: R -> String
showDouble x
    | x < 0      = "(" ++ show x ++ ")"
    | otherwise  = show x

-- | Form a vector by giving its x, y, and z components.
vec :: R  -- ^ x component
    -> R  -- ^ y component
    -> R  -- ^ z component
    -> Vec
```

```
vec = Vec
```

Here are unit vectors in the x, y, and z directions.

```
iHat :: Vec
iHat = vec 1 0 0

jHat :: Vec
jHat = vec 0 1 0

kHat :: Vec
kHat = vec 0 0 1
```

```
-- | The zero vector.
zeroV :: Vec
zeroV = vec 0 0 0

-- | The additive inverse of a vector.
negateV :: Vec -> Vec
negateV (Vec ax ay az) = Vec (-ax) (-ay) (-az)
```

Vector addition and subtraction are just the addition and subtraction of the corresponding Cartesian components.

```
(^+^) :: Vec -> Vec -> Vec
Vec ax ay az ^+^ Vec bx by bz = Vec (ax+bx) (ay+by) (az+bz)

(^-^) :: Vec -> Vec -> Vec
Vec ax ay az ^-^ Vec bx by bz = Vec (ax-bx) (ay-by) (az-bz)
```

It is useful to have a function that adds a whole list of vectors. We will use this function when we do numeric integrals.

```
sumV :: [Vec] -> Vec
sumV = foldr (^+^) zeroV
```

The function `foldr` is defined in the prelude. The definition of `sumV` is written in point-free style, which means that it is short for `sumV vs = foldr (^+^) zeroV vs`. Roughly speaking, `foldr` takes a binary operator (`(^+^)` in this case), an initial value, and a list of values, and "folds" the initial value and an element from the list into an accumulated value, then continues to fold the accumulate value with the next element of this to form a new accumulated value, until the list is gone and the final accumulated value is returned. It is a fairly powerful function, but here it is used to just keep adding the members of the list until there are no more.

There are three types of multiplication in which three-dimensional vectors participate. The first is scalar multiplication, in which we multiply a number by a vector or a vector by a number. We use `(*^)` and `(^*)` for scalar multiplication. The first takes a number on the left and a vector on the right. The second takes a vector on the left and a number on the right. The vector always goes next to the carrot symbol. The second vector multiplication is the dot product. We use `(<.>)` for the dot product. The third vector multiplication is the cross product. We use `(><)` for the cross product, because `><` looks a bit like a cross.

```
(*^)  :: R   -> Vec -> Vec
c *^ Vec ax ay az = Vec (c*ax) (c*ay) (c*az)

(^*)  :: Vec -> R   -> Vec
Vec ax ay az ^* c = Vec (c*ax) (c*ay) (c*az)

(<.>) :: Vec -> Vec -> R
Vec ax ay az <.> Vec bx by bz = ax*bx + ay*by + az*bz

(><)  :: Vec -> Vec -> Vec
Vec ax ay az >< Vec bx by bz
    = Vec (ay*bz - az*by) (az*bx - ax*bz) (ax*by - ay*bx)
```

We can divide a vector by a scalar.

```
(^/) :: Vec -> R -> Vec
Vec ax ay az ^/ c = Vec (ax/c) (ay/c) (az/c)
```

We can take the magnitude of a vector.

```
magnitude :: Vec -> R
magnitude v = sqrt(v <.> v)
```

Appendix C

Vector Integrals

C.1 A table of vector integrals

scalar line integral	$\int f\,dl$	1D
vector line integral	$\int \mathbf{F}\,dl$	1D
dotted line integral	$\int \mathbf{F}\cdot d\mathbf{l}$	1D
scalar surface integral	$\int f\,da$	2D
vector surface integral	$\int \mathbf{F}\,da$	2D
flux integral	$\int \mathbf{F}\cdot d\mathbf{a}$	2D
scalar volume integral	$\int f\,dv$	3D
vector volume integral	$\int \mathbf{F}\,dv$	3D

C.2 Applications of the integrals

C.2.1 Scalar line integral

The scalar line integral requires a scalar field f and a path P.

$$\int_P f\,dl$$

C.2.1.1 Finding total charge of a line charge

The path P is along the line charge.

$$Q = \int_P \lambda(\mathbf{r}')\,dl'$$

C.2.1.2 Finding electric potential of a line charge

The path P is along the line charge.

$$\phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int_P \frac{\lambda(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} \, dl'$$

C.2.2 Vector line integral

The vector line integral requires a vector field $\mathbf{F}$ and a path P.

$$\int_P \mathbf{F} \, dl$$

C.2.2.1 Finding electric field of a line charge

The path P is along the line charge.

$$\mathbf{E}(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int_P \frac{\mathbf{r} - \mathbf{r}'}{|\mathbf{r} - \mathbf{r}'|^3} \lambda(\mathbf{r}') \, dl'$$

C.2.3 Dotted line integral

The dotted line integral requires a vector field $\mathbf{F}$ and a path P.

$$\int_P \mathbf{F} \cdot d\mathbf{l}$$

C.2.3.1 Finding electric potential from electric field

The path P begins at a point where the electric potential is zero and ends at the field point $\mathbf{r}$.

$$\phi(\mathbf{r}) = -\int_P \mathbf{E}(\mathbf{r}') \cdot d\mathbf{l}'$$

In electrostatics, this integral is path independent, so we could write the integral using only the endpoints of the path P. Let $\mathbf{a}_0$ be a point we choose for the electric potential to be zero.

$$\phi(\mathbf{r}) = -\int_{\mathbf{a}_0}^{\mathbf{r}} \mathbf{E}(\mathbf{r}') \cdot d\mathbf{l}'$$

A nice property of this latter form is that we see how the field point $\mathbf{r}$ on the left is related to the field point $\mathbf{r}$ on the right. We are finding the electric potential at the end point $\mathbf{r}$ of the path P.

C.2.4 Scalar surface integral

The scalar surface integral requires a scalar field f and a surface S. The surface does not need an orientation.

$$\int_S f\,da$$

C.2.4.1 Finding total charge of a surface charge

The surface S is over the surface charge.

$$Q = \int_S \sigma(\mathbf{r}')\,da'$$

C.2.4.2 Finding electric potential of a surface charge

The surface S is over the surface charge.

$$\phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\int_S \frac{\sigma(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}\,da'$$

C.2.5 Vector surface integral

The vector surface integral requires a vector field $\mathbf{F}$ and a surface S. The surface does not need an orientation.

$$\int_S \mathbf{F}\,da$$

C.2.5.1 Finding electric field of a surface charge

The surface S is over the surface charge.

$$\mathbf{E}(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\int_S \frac{\mathbf{r}-\mathbf{r}'}{|\mathbf{r}-\mathbf{r}'|^3}\sigma(\mathbf{r}')\,da'$$

C.2.6 Flux integral

The flux integral requires a vector field $\mathbf{F}$ and an oriented surface S.

$$\int_S \mathbf{F}\cdot d\mathbf{a}$$

C.2.6.1 Finding electric flux from electric field

The surface S is the surface through which to find the electric flux. The orientation points perpendicular to the surface, so that electric field in that direction would count as positive electric flux.

$$\Phi_E = \int_S \mathbf{E}(\mathbf{r}') \cdot d\mathbf{a}'$$

C.2.7 Scalar volume integral

The scalar volume integral requires a scalar field f and a volume V.

$$\int_V f \, dv$$

C.2.7.1 Finding total charge of a volume charge

The volume V is over the volume charge.

$$Q = \int_V \rho(\mathbf{r}') \, dv'$$

C.2.7.2 Finding electric potential of a volume charge

The volume V is over the volume charge.

$$\phi(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int_V \frac{\rho(\mathbf{r}')}{|\mathbf{r} - \mathbf{r}'|} \, dv'$$

C.2.8 Vector volume integral

The vector volume integral requires a vector field $\mathbf{F}$ and a volume V.

$$\int_V \mathbf{F} \, dv$$

C.2.8.1 Finding electric field of a volume charge

The volume V is over the volume charge.

$$\mathbf{E}(\mathbf{r}) = \frac{1}{4\pi\epsilon_0} \int_V \frac{\mathbf{r} - \mathbf{r}'}{|\mathbf{r} - \mathbf{r}'|^3} \rho(\mathbf{r}') \, dv'$$

C.3 Code for integrals

As usual, we will construct a Haskell module in this chapter, and we will start by turning on warnings, giving our module the name `VectorIntegrals`, and importing some functions that we will need later in the chapter.

```
{-# OPTIONS_GHC -Wall #-}

module VectorIntegrals where

import CoordinateSystems
    ( R
    , Position
    , ScalarField
    , VectorField
    , cartesianCoordinates
    , cart
    , displacement
    )
import Geometry
    ( Curve(..)
    , Surface(..)
    , Volume(..)
    )
import SimpleVec
    ( Vec
    , magnitude
    , sumV
    , (^+^)
    , (^*)
    , (^/)
    , (<.>)
    , (><)
    )
```

Above I have explicitly listed the types and functions I want to import from other modules to use here. When we include the type `Curve` followed by two dots `(..)` in parentheses, it means that we want to import the `Curve` type as well as all of the data constructors for `Curve` (of which there is only one, called `Curve`).

C.3.1 Scalar line integral

The scalar line integral requires a scalar field f and a path or curve (I use the words *path* and *curve* interchangeably) P.

$$\int_P f\, dl$$

We get a scalar as a result. Our mission, then, is to write a function that looks something like the following, with `undefined` replaced by whatever the scalar line integral really means. (I have put a `1` at the end of `scalarLineIntegral` because I think the type will actually change a little bit as we become clear about what we want the function to do for us.)

```
scalarLineIntegral1 :: ScalarField -> Curve -> R
scalarLineIntegral1 = undefined
```

What does the scalar line integral mean? What do we want this code to do for us? Let's take a look at these questions by way of an example. For our scalar field, let's choose $f(x, y, z) = x^2 + y^2$. For our curve, let's choose the parabola

$$t \mapsto (t, t^2, 0)$$

in which $0 \leq t \leq 2$. Here are our example scalar field and curve in Haskell.

```
fsf :: ScalarField
fsf p = let (x,y,_) = cartesianCoordinates p
        in x*x + y*y

parabola :: Curve
parabola = Curve (\t -> cart t (t*t) 0) 0 2
```

A line integral is *adding something up* along a curve. When we numerically integrate, we take a continuous thing (the original integral) and convert it into a discrete thing (a sum). In the case of a line integral, we divide our curve into some finite number of sections or segments. For our current example and our pictures, we will choose $N = 4$ segments. This will make it easier on the brain to understand what we are doing, and easier to read the pictures. (If we want the numerical integral to be a reasonable approximation to the original continous integral, we should probably choose $N = 100$ or $N = 1000$.)

The range of parameters for our curve is the closed interval $[0, 2]$. We will break this parameter space into four equal parts. The four parameter sections are the intervals

$$[0.0, 0.5], [0.5, 1.0], [1.0, 1.5], [1.5, 2.0].$$

We identify the points at the beginning and end of each interval. There will be $N + 1 = 5$ such points at $t = 0.0, 0.5, 1.0, 1.5, 2.0$. Our curve, divided into four sections, is shown in Figure C.1.

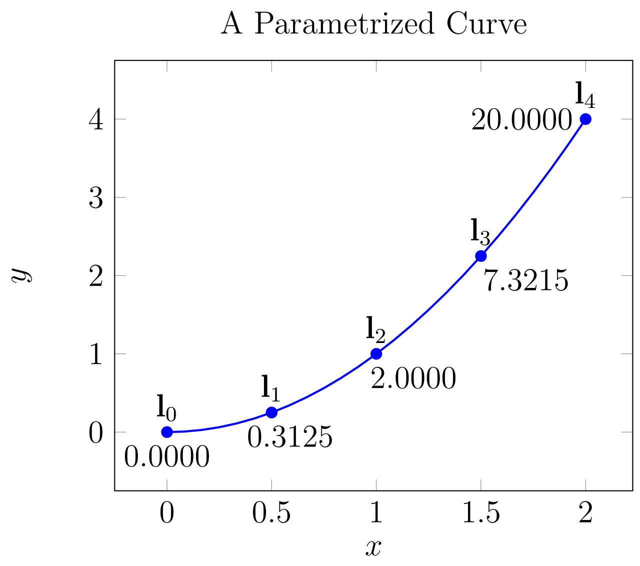
Figure C.1 shows the end points of our curve when we divide the curve into four segments. How can we find these end points for any curve with any N? Here is a function to do that.

```haskell
sectionEndPoints :: Int          -- ^ number of intervals
                 -> Curve
                 -> [Position]   -- ^ list of endpoints
sectionEndPoints n (Curve g a b)
    = let dt = (b - a) / fromIntegral n
      in [g t | t <- [a, a + dt .. b]]
```

We give the function `sectionEndPoints` a number of intervals N (called `n` in the code) along with a curve, and the function gives us back a list of positions of the $N + 1$ end points. Th function works by first finding out the parameter distance `dt` between adjacent end points. For this, we just need to divide the entire parameter region (`b - a`) by N. Since `n` has type `Int` and `b - a` has type `R`, we need the `fromIntegral` function to convert `n` to type `R` before we do the division. The list comprehension in the last line applies the curve function `g` to each parameter in the full list of parameters that starts with `a`, increases by `dt`, and ends at `b`.

Let's confirm that GHCi gives the correct value for $\mathbf{l}_3$ in Figure C.1.

```
GHCi  :l VectorIntegrals.lhs
```



Parameter t	End point $\mathbf{l}$	Scalar field value
$t_0 = 0.0$	$\mathbf{l}_0 = (0.00, 0.00, 0.00)$	$f(\mathbf{l}_0) = 0.0000$
$t_1 = 0.5$	$\mathbf{l}_1 = (0.50, 0.25, 0.00)$	$f(\mathbf{l}_1) = 0.3125$
$t_2 = 1.0$	$\mathbf{l}_2 = (1.00, 1.00, 0.00)$	$f(\mathbf{l}_2) = 2.0000$
$t_3 = 1.5$	$\mathbf{l}_3 = (1.50, 2.25, 0.00)$	$f(\mathbf{l}_3) = 7.3215$
$t_4 = 2.0$	$\mathbf{l}_4 = (2.00, 4.00, 0.00)$	$f(\mathbf{l}_4) = 20.0000$

Figure C.1: The parametrized curve $t \mapsto (t, t^2, 0)$ divided into $N = 4$ equal sections. The values of the scalar field $f(x, y, z) = x^2 + y^2$ are shown at each end point.

GHCi
```
sectionEndPoints 4 parabola !! 3
    ⇝  Cart 1.5 2.25 0.0
```

The other information shown in Figure C.1 is the value of the scalar field f at the curve end points. How can we find these values in general?

```
fVals :: Int     -- ^ number of intervals
      -> ScalarField
      -> Curve
      -> [R]
fVals n f c = [f pt | pt <- sectionEndPoints n c]
```

The `fVals` function takes a number of intervals, a scalar field, and a curve, and produces a list of values that the scalar field takes at each of the $N+1$ end points. We see that the function uses a list comprehension to apply the scalar field `f` to each end point in the list formed with the `sectionEndPoints` function.

GHCi
```
fVals 4 fsf parabola
    ⇝  [0.0,0.3125,2.0,7.3125,20.0]
```

The numerical approximation we make is

$$\int_P f\,dl \approx \sum_{j=1}^{N} f_j \Delta l_j, \tag{C.1}$$

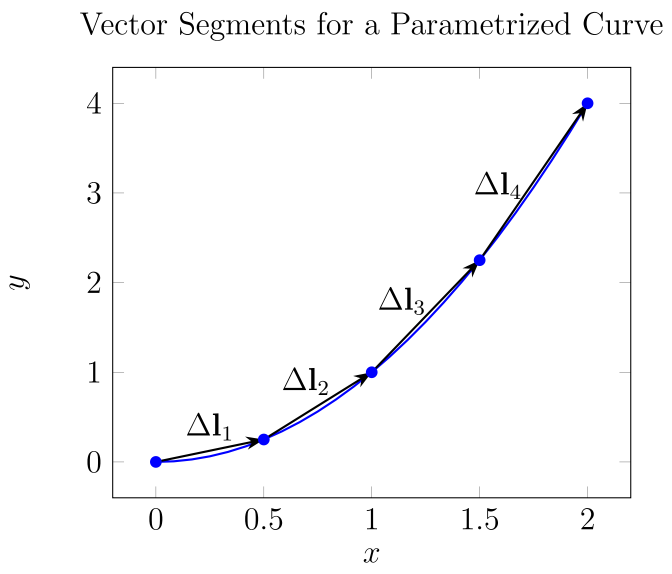
where Δl_j is the length of a section of our curve P and f_j is the value of the scalar field f on or near that section. After we compute the sum, we are left with a scalar value for the scalar line integral.

We have some choices to make in equation C.1 before we have really been precise about what the computer is supposed to do.

(a) We must decide how to approximate the length Δl_j of each section of the curve.

(b) We must decide what value f_j of the scalar field to use for each section of the curve.

We then multiply each length by the corresponding value, and add it up.



Vector segment $\Delta\mathbf{l}$	Segment length Δl
$\Delta\mathbf{l}_1 = (0.50, 0.25, 0.00)$	$\Delta l_1 = 0.5590$
$\Delta\mathbf{l}_2 = (0.50, 0.75, 0.00)$	$\Delta l_2 = 0.9014$
$\Delta\mathbf{l}_3 = (0.50, 1.25, 0.00)$	$\Delta l_3 = 1.3463$
$\Delta\mathbf{l}_4 = (0.50, 1.75, 0.00)$	$\Delta l_4 = 1.8200$

Figure C.2: The vector segments for the curve $t \mapsto (t, t^2, 0)$ when $N = 4$.

How shall we approximate the length of a curve section? There are different options, but one simple way is to use the length of the vector that joins the endpoints of a curve section. See Figure C.2.

We need a general purpose way to find the vector segments listed in Figure C.2.

```
dlVecs :: Int     -- ^ number of intervals
       -> Curve
       -> [Vec]   -- ^ list of dl vectors
dlVecs n c = let pts = sectionEndPoints n c
             in zipWith displacement pts (tail pts)
```

This function takes a number N of intervals along with a curve and returns a list of N vector segments along the curve. The function works by first finding the list of $N + 1$ section end points and storing them in the local list variable `pts`. The trick in the second line is to use `zipWith displacement` to find the displacement from the first item in `pts` to the first item in `tail pts`, the displacement from the second item in `pts` to the second item in `tail pts`, and so on. The list `pts` has $N + 1$ items; the list `tail pts` has N items. When using `zip` or `zipWith`, the resulting list has the length of the shorter list, so the final list of vector segments has length N.

Let's confirm that GHCi gives the correct value for $\Delta\mathbf{l}_4$ in Figure C.2.

```
GHCi   dlVecs 4 parabola !! 3
           ⇝  vec 0.5 1.75 0.0
```

Note that we ask for element 3, which is really the fourth element, since we start counting our elements from zero. The length of this vector can be obtained with `magnitude`.

```
GHCi   magnitude (dlVecs 4 parabola !! 3)
           ⇝  1.8200274723201295
```

We are almost there. For our example scalar line integral, we have

$$\begin{aligned}\int_P f\,dl &\approx \sum_{j=1}^{N} f_j \Delta l_j \\ &= f_1\Delta l_1 + f_2\Delta l_2 + f_3\Delta l_3 + f_4\Delta l_4 \\ &= 0.5590 f_1 + 0.9014 f_2 + 1.3463 f_3 + 1.8200 f_4\end{aligned}$$

We must now make decision (b) and figure out how to get values f_j of the scalar field on or near each curve segment.

We have the five segment end points $\mathbf{l}_0, \dots, \mathbf{l}_4$ which we used to get values of the scalar field. For example, when we evaluate the scalar field f (called `fsf` in the Haskell code) at the end point $\mathbf{l}_3 = (1.50, 2.25, 0.00)$,

GHCi
```
fsf (cart 1.50 2.25 0.00)
    ⇝  7.3125
```

we get a number. The trouble in using this number for f_3, say, is that the number doesn't really go with *segment* 3; it goes with the point between segment 3 and segment 4.

(1) We could find points in space for parameters $t = 0.25, 0.75, 1.25, 1.75$ and then evaluate the scalar field at these space points. These points are on the curve, and near the middle of the segments, so this is reasonable.

(2) We could evaluate the 5 points we already have, and average the values.

We will go with option (2). The averaging can be done with the following function.

```
averageVals :: [R] -> [R]
averageVals vs = [(x + y) / 2 | (x,y) <- zip vs (tail vs)]
```

We are now ready to give our Haskell expression for a scalar line integral.

```
scalarLineIntegral
    :: Int            -- ^ number of intervals
    -> ScalarField    -- ^ scalar field
    -> Curve          -- ^ curve to integrate over
    -> R              -- ^ scalar result
scalarLineIntegral n f c
    = sum $ zipWith (*) segmentVals (map magnitude dls)
      where
        segmentVals = averageVals (fVals n f c)
        dls         = dlVecs n c
```

GHCi
```
scalarLineIntegral 4 fsf parabola
   ⇝ 32.25299464811167
```
GHCi
```
scalarLineIntegral 100 fsf parabola
   ⇝ 30.778970402732043
```
GHCi
```
scalarLineIntegral 1000 fsf parabola
   ⇝ 30.776568015034147
```

C.3.2 Vector line integral

```
vectorLineIntegral
    :: Int            -- ^ number of intervals
    -> VectorField    -- ^ vector field
    -> Curve          -- ^ curve to integrate over
    -> Vec            -- ^ vector result
vectorLineIntegral n vf c
    = sumV $ zipWith (^*) segmentVals (map magnitude dls)
      where
        segmentVals = averageVFVals (vfVals n vf c)
        dls         = dlVecs n c

vfVals :: Int     -- ^ number of intervals
       -> VectorField
       -> Curve
       -> [Vec]
vfVals n vf c = [vf pt | pt <- sectionEndPoints n c]

averageVFVals :: [Vec] -> [Vec]
averageVFVals vs = [(x ^+^ y) ^/ 2 | (x,y) <- zip vs (tail vs)]
```

C.3.3 Dotted line integral

```
dottedLineIntegral
    :: Int            -- ^ number of intervals
    -> VectorField    -- ^ vector field
    -> Curve         -- ^ curve to integrate over
    -> R             -- ^ scalar result
dottedLineIntegral n vf c
    = sum $ zipWith (<.>) segmentVals dls
      where
        segmentVals = averageVFVals (vfVals n vf c)
        dls         = dlVecs n c
```

C.3.3.1 Comparison of line integrals

All three of our line integrals take a curve and a number of intervals as inputs. Beyond that, the types differ. The following table gives a comparison of the differences.

Line integral	f or $\mathbf{F}$ values	combinator	dl or $d\mathbf{l}$ values	Output
`scalarLineIntegral`	`R`	`(*)`	`R`	`R`
`vectorLineIntegral`	`Vec`	`(^*)`	`R`	`Vec`
`dottedLineIntegral`	`Vec`	`(<.>)`	`Vec`	`R`

C.3.4 Scalar surface integral

```
linSpaced :: Int -> R -> R -> [R]
linSpaced n a b
    | abs (a - b) < tolerance  = [(a + b)/2]
    | otherwise                = let dx = (b - a) / fromIntegral n
                                 in [a,a+dx..b]

tolerance :: R
tolerance = 1e-10

ave :: R -> R -> R
ave v1 v2 = (v1 + v2) / 2

aveV :: Vec -> Vec -> Vec
aveV v1 v2 = (v1 ^+^ v2) ^/ 2
```

```
scalarSurfaceIntegral :: Int          -- ^ number of intervals for first parameter, s
                      -> Int          -- ^ number of intervals for second parameter,
                      -> ScalarField  -- ^ the scalar or vector field to integrate
                      -> Surface      -- ^ the surface over which to integrate
                      -> R            -- ^ the resulting scalar or vector
scalarSurfaceIntegral n1 n2 field (Surface f s_l s_u t_l t_u)
    = sum $ map sum $ zipWith (zipWith (*)) aveVals (map (map magnitude) areas)
      where
        pts = [[f (s,t) | t <- linSpaced n2 (t_l s) (t_u s)] | s <- linSpaced n1 s_l
        areas = zipWith (zipWith (><)) dus dvs
        dus = zipWith (zipWith displacement) pts (tail pts)
        dvs = map (\row -> zipWith displacement row (tail row)) pts
        vals = map (map field) pts
        halfAveVals = map (\row -> zipWith ave (tail row) row) vals
        aveVals = zipWith (zipWith ave) (tail halfAveVals) halfAveVals
```

C.3.5 Vector surface integral

```
vectorSurfaceIntegral :: Int          -- ^ number of intervals for first parameter, s
                      -> Int          -- ^ number of intervals for second parameter,
                      -> VectorField  -- ^ the scalar or vector field to integrate
                      -> Surface      -- ^ the surface over which to integrate
                      -> Vec          -- ^ the resulting scalar or vector
vectorSurfaceIntegral n1 n2 field (Surface f s_l s_u t_l t_u)
    = sumV $ map sumV $ zipWith (zipWith (^*)) aveVals (map (map magnitude) areas)
      where
        pts = [[f (s,t) | t <- linSpaced n2 (t_l s) (t_u s)] | s <- linSpaced n1 s_l
        areas = zipWith (zipWith (><)) dus dvs
        dus = zipWith (zipWith displacement) pts (tail pts)
        dvs = map (\row -> zipWith displacement row (tail row)) pts
        vals = map (map field) pts
        halfAveVals = map (\row -> zipWith aveV (tail row) row) vals
        aveVals = zipWith (zipWith aveV) (tail halfAveVals) halfAveVals
```

C.3.6 Flux integral

```
dottedSurfaceIntegral :: Int           -- ^ number of intervals for first param
                      -> Int           -- ^ number of intervals for second para
                      -> VectorField   -- ^ the vector field to integrate
                      -> Surface       -- ^ the surface over which to integrate
                      -> R             -- ^ the resulting scalar
dottedSurfaceIntegral n1 n2 vf (Surface f s_l s_u t_l t_u)
    = sum $ map sum $ zipWith (zipWith (<.>)) aveVals areas
      where
        pts = [[f (s,t) | t <- linSpaced n2 (t_l s) (t_u s)] | s <- linSpaced
        areas = zipWith (zipWith (><)) dus dvs
        dus = zipWith (zipWith displacement) pts (tail pts)
        dvs = map (\row -> zipWith displacement row (tail row)) pts
        vals = map (map vf) pts
        halfAveVals = map (\row -> zipWith aveV (tail row) row) vals
        aveVals = zipWith (zipWith aveV) (tail halfAveVals) halfAveVals
```

C.3.7 Scalar volume integral

```
zipCubeWith :: (a -> b -> c) -> [[[a]]] -> [[[b]]] -> [[[c]]]
zipCubeWith = zipWith . zipWith . zipWith

zipSub3 :: [[[Position]]] -> [[[Position]]] -> [[[Vec]]]
zipSub3 = zipCubeWith displacement

zipAve3 :: [[[R]]] -> [[[R]]] -> [[[R]]]
zipAve3 = zipCubeWith ave

zipAve3V :: [[[Vec]]] -> [[[Vec]]] -> [[[Vec]]]
zipAve3V = zipCubeWith aveV

shift1 :: [a] -> ([a],[a])
shift1 pts = (pts, tail pts)

shift2 :: [[a]] -> ([[a]],[[a]])
```

```
shift2 pts2d = (pts2d, map tail pts2d)

shift3 :: [[[a]]] -> ([[[a]]],[[[a]]])
shift3 pts3d = (pts3d, map (map tail) pts3d)


scalarVolumeIntegral :: Int          -- ^ number of intervals for first parameter   (
                     -> Int          -- ^ number of intervals for second parameter  (
                     -> Int          -- ^ number of intervals for third parameter   (
                     -> ScalarField  -- ^ scalar or vector field
                     -> Volume       -- ^ the volume
                     -> R            -- ^ scalar or vector result
scalarVolumeIntegral n1 n2 n3 field (Volume f s_l s_u t_l t_u u_l u_u)
    = sum $ map sum $ map (map sum) (zipCubeWith (*) aveVals volumes)
      where
        pts = [[[f (s,t,u) | u <- linSpaced n3 (u_l s t) (u_u s t) ] | t <- linSpaced
        volumes = zipWith3 (zipWith3 (zipWith3 (\du dv dw -> du <.> (dv >< dw)))) dus
        dus = uncurry zipSub3 (shift1 pts)
        dvs = uncurry zipSub3 (shift2 pts)
        dws = uncurry zipSub3 (shift3 pts)
        vals = map (map (map field)) pts
        aveVals = ((uncurry zipAve3 . shift1) . (uncurry zipAve3 . shift2) . (uncurry
```

C.3.8 Vector volume integral

```
vectorVolumeIntegral :: Int          -- ^ number of intervals for first parameter   (
                     -> Int          -- ^ number of intervals for second parameter  (
                     -> Int          -- ^ number of intervals for third parameter   (
                     -> VectorField  -- ^ scalar or vector field
                     -> Volume       -- ^ the volume
                     -> Vec          -- ^ scalar or vector result
vectorVolumeIntegral n1 n2 n3 field (Volume f s_l s_u t_l t_u u_l u_u)
    = sumV $ map sumV $ map (map sumV) (zipCubeWith (^*) aveVals volumes)
      where
        pts = [[[f (s,t,u) | u <- linSpaced n3 (u_l s t) (u_u s t) ] | t <- linSpaced
        volumes = zipWith3 (zipWith3 (zipWith3 (\du dv dw -> du <.> (dv >< dw)))) dus
        dus = uncurry zipSub3 (shift1 pts)
        dvs = uncurry zipSub3 (shift2 pts)
        dws = uncurry zipSub3 (shift3 pts)
        vals = map (map (map field)) pts
        aveVals = ((uncurry zipAve3V . shift1) . (uncurry zipAve3V . shift2) .
```

C.4 Fundamental theorems of calculus

C.4.1 Gradient theorem

The gradient theorem requires a scalar field f and a path P. Suppose the path P starts at $\mathbf{a}$ and ends at $\mathbf{b}$.

$$\int_P \nabla f \cdot d\mathbf{l} = f(\mathbf{b}) - f(\mathbf{a})$$

The integral on the left is a dotted line integral.

C.4.2 Stokes' theorem

Stokes' theorem requires a vector field $\mathbf{F}$ and an oriented surface S. Let ∂S be a (closed) path that forms the boundary of the surface S.

$$\int_S (\nabla \times \mathbf{F}) \cdot d\mathbf{a} = \int_{\partial S} \mathbf{F} \cdot d\mathbf{l}$$

The integral on the left is a flux integral, and the integral on the right is a dotted line integral.

C.4.3 Divergence theorem

The divergence theorem requires a vector field $\mathbf{F}$ and a volume V. Let ∂V be the (closed) oriented surface that forms the boundary of the volume V. The orientation is "outward".

$$\int_V (\nabla \cdot \mathbf{F})\, dv = \int_{\partial V} \mathbf{F} \cdot d\mathbf{a}$$

The integral on the left is a scalar volume integral, and the integral on the right is a flux integral.

C.5 Calculation

C.5.1 Line integrals

The key to evaluating line integrals, whether they be scalar line integrals, vector line integrals, or dotted line integrals, is to find a single variable or parameter in which to carry out the integration. In many cases, this single parameter can be one of the coordinates in a Cartesion, cylindrical, or spherical coordinate system.

C.5.1.1 Scalar line integrals

Example C.1. Find the scalar line integral $\int_P f\,dl$ for the scalar field

$$f(x,y,z) = x^2 + y^2$$

over the path running along the parabola $t \mapsto (t, t^2, 0)$ from $t = 0$ to $t = 2$.

Solution:

$$\mathbf{l}(t) = (t, t^2, 0)$$

$$f = x^2 + y^2 = t^2 + t^4$$

$$d\mathbf{l} = (1, 2t, 0)\,dt$$

$$dl = \sqrt{1+4t^2}\,dt$$

$$\begin{aligned}\int_P f\,dl &= \int_0^2 (t^2+t^4)\sqrt{1+4t^2}\,dt \\ &= \frac{11476\sqrt{17} - 21\,\mathrm{asinh}(4)}{1536} \\ &= 30.7765\end{aligned}$$

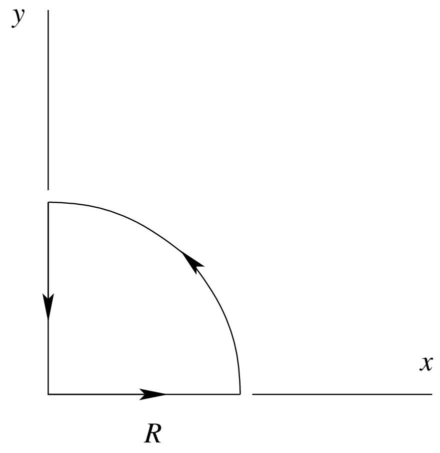
(Using Maxima to calculate the second line.) We got 30.78 from our numerical integrals.

C.5.1.2 Vector line integrals

Example C.2. Find the vector line integral $\int \mathbf{F}\,dl$ for the vector field

$$\mathbf{F}(s,\phi,z) = s^2\cos\phi\hat{\mathbf{s}} + s^2\sin\phi\hat{\boldsymbol{\phi}}$$

over the closed path shown below.



Solution: The path is made up of three portions. Let P_1 be the portion that runs along the x axis. Let P_2 be the portion that curves, and let P_3 be the portion that runs along the y axis. Denote by P the entire closed path. The vector line integral over the entire path is the sum of the vector line integrals over each portion.

$$\int_P \mathbf{F}\,dl = \int_{P_1} \mathbf{F}\,dl + \int_{P_2} \mathbf{F}\,dl + \int_{P_3} \mathbf{F}\,dl$$

We will carry the integrals out in cylindrical coodinates. However, we want to avoid having the unit vectors $\hat{\mathbf{s}}$ and $\hat{\boldsymbol{\phi}}$ under the integral sign, because these unit vectors change direction from place to place. Therefore, we will rewrite the vector field $\mathbf{F}$ using Cartesian unit vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$, but keeping the cylindrical coordinates s, ϕ, and z.

We can write the cylindrical coordinate unit vectors $\hat{\mathbf{s}}$, $\hat{\boldsymbol{\phi}}$, and $\hat{\mathbf{z}}$ in terms of the Cartesian coordinate unit vectors $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ as follows.

$$\begin{aligned}
\hat{\mathbf{s}} &= \cos\phi\hat{\mathbf{x}} + \sin\phi\hat{\mathbf{y}} \\
\hat{\boldsymbol{\phi}} &= -\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}} \\
\hat{\mathbf{z}} &= \hat{\mathbf{z}}
\end{aligned}$$

The vector field $\mathbf{F}$ is the following.

$$\begin{aligned}\mathbf{F} &= s^2\cos\phi\hat{\mathbf{s}} + s^2\sin\phi\hat{\boldsymbol{\phi}}\\ &= s^2\cos\phi(\cos\phi\hat{\mathbf{x}} + \sin\phi\hat{\mathbf{y}}) + s^2\sin\phi(-\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}})\\ &= s^2\cos 2\phi\hat{\mathbf{x}} + s^2\sin 2\phi\hat{\mathbf{y}}\end{aligned}$$

Here, we have used the following trigonometric identities.

$$\begin{aligned}\cos 2\theta &= \cos^2\theta - \sin^2\theta\\ \sin 2\theta &= 2\sin\theta\cos\theta\end{aligned}$$

For the first portion P_1, we have $dl = ds$.

$$\int_{P_1}\mathbf{F}\,dl = \int_0^R (s^2\cos 2\phi\hat{\mathbf{x}} + s^2\sin 2\phi\hat{\mathbf{y}})\,ds$$

We want to get all variables in terms of s, since that is the one variable we are integrating over. Along P_1, we have $\phi = 0$.

$$\int_{P_1}\mathbf{F}\,dl = \hat{\mathbf{x}}\int_0^R s^2\,ds = \frac{R^3}{3}\hat{\mathbf{x}}$$

For the second portion P_2, we have $dl = s\,d\phi$.

$$\int_{P_2}\mathbf{F}\,dl = \int_0^{\pi/2}(s^2\cos 2\phi\hat{\mathbf{x}} + s^2\sin 2\phi\hat{\mathbf{y}})s\,d\phi$$

Along P_2, we have $s = R$.

$$\begin{aligned}\int_{P_2}\mathbf{F}\,dl &= R^3\int_0^{\pi/2}(\cos 2\phi\hat{\mathbf{x}} + \sin 2\phi\hat{\mathbf{y}})\,d\phi\\ &= R^3\hat{\mathbf{x}}\int_0^{\pi/2}\cos 2\phi\,d\phi + R^3\hat{\mathbf{y}}\int_0^{\pi/2}\sin 2\phi\,d\phi\end{aligned}$$

$$\int_0^{\pi/2}\cos 2\phi\,d\phi = \left[\frac{\sin 2\phi}{2}\right]_0^{\pi/2} = 0$$

$$\int_0^{\pi/2}\sin 2\phi\,d\phi = \left[-\frac{\cos 2\phi}{2}\right]_0^{\pi/2} = \left[\frac{\cos 2\phi}{2}\right]_{\pi/2}^0 = \frac{1-(-1)}{2} = 1$$

$$\int_{P_2} \mathbf{F}\, dl = R^3 \hat{\mathbf{y}}$$

For the third portion P_3, we have $dl = ds$.

$$\int_{P_3} \mathbf{F}\, dl = \int_R^0 (s^2 \cos 2\phi \hat{\mathbf{x}} + s^2 \sin 2\phi \hat{\mathbf{y}})\, ds$$

This looks very much like the expression for the integral over the first portion above, but it's different in two ways. First, the limits on the integral are different. For P_3, we're starting at $s = R$ and going to $s = 0$. Second, the value of ϕ is different. For P_3, we have $\phi = \pi/2$.

$$\int_{P_3} \mathbf{F}\, dl = -\hat{\mathbf{x}} \int_R^0 s^2\, ds = \frac{R^3}{3} \hat{\mathbf{x}}$$

In total we have the following result.

$$\int_P \mathbf{F}\, dl = \frac{2}{3} R^3 \hat{\mathbf{x}} + R^3 \hat{\mathbf{y}}$$

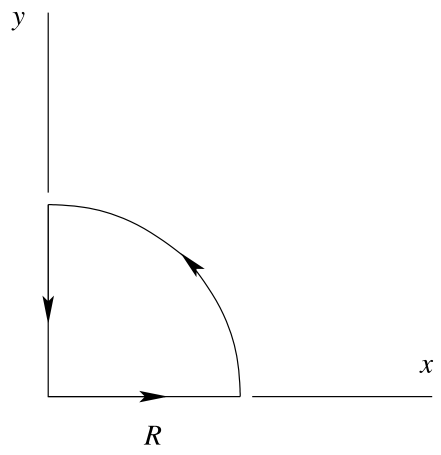
Note that the result of a vector line integral is a vector.

C.5.1.3 Dotted line integrals

Example C.3. Find the dotted line integral $\int \mathbf{F} \cdot d\mathbf{l}$ for the vector field

$$\mathbf{F}(s, \phi, z) = s^2 \cos \phi \hat{\mathbf{s}} + s^2 \sin \phi \hat{\boldsymbol{\phi}}$$

over the closed path shown below.



Solution: The path is made up of three portions. Let P_1 be the portion that runs along the x axis. Let P_2 be the portion that curves, and let P_3 be the portion that runs along the y axis. Denote by P the entire closed path. The dotted line integral over the entire path is the sum of the dotted line integrals over each portion.

$$\int_P \mathbf{F} \cdot d\mathbf{l} = \int_{P_1} \mathbf{F} \cdot d\mathbf{l} + \int_{P_2} \mathbf{F} \cdot d\mathbf{l} + \int_{P_3} \mathbf{F} \cdot d\mathbf{l}$$

We will carry the integrals out in cylindrical coodinates. For $d\mathbf{l}$, we can use the standard expression in cylindrical coordinates.

$$d\mathbf{l} = ds\hat{\mathbf{s}} + s\,d\phi\hat{\boldsymbol{\phi}} + dz\hat{\mathbf{z}}$$

Now we can find an expression for $\mathbf{F} \cdot d\mathbf{l}$.

$$\begin{aligned}\mathbf{F} \cdot d\mathbf{l} &= (s^2\cos\phi\hat{\mathbf{s}} + s^2\sin\phi\hat{\boldsymbol{\phi}}) \cdot (ds\hat{\mathbf{s}} + s\,d\phi\hat{\boldsymbol{\phi}} + dz\hat{\mathbf{z}}) \\ &= s^2\cos\phi\,ds + s^3\sin\phi\,d\phi\end{aligned}$$

Recall that for line integrals of all kinds, we want to express the integral in terms of a single variable that we are integrating over. The expression above is not yet in that form. We have three portions of the path that we want to integrate over, and it will turn out that the variable we want for our single variable of integration is different for the different portions. The expression for $\mathbf{F} \cdot d\mathbf{l}$ will simplify in different ways for each portion of the path.

For the first portion P_1, we have $\phi = 0$. This means that $d\phi = 0$ and that part of $\mathbf{F} \cdot d\mathbf{l}$ will go away. For P_1, we have

$$\mathbf{F} \cdot d\mathbf{l} = s^2\,ds.$$

$$\int_{P_1} \mathbf{F} \cdot d\mathbf{l} = \int_0^R s^2\,ds = \frac{R^3}{3}$$

For the second portion P_2, we have $s = R$, $ds = 0$, and

$$\mathbf{F} \cdot d\mathbf{l} = R^3\sin\phi\,d\phi.$$

$$\int_{P_2} \mathbf{F} \cdot d\mathbf{l} = \int_0^{\pi/2} R^3\sin\phi\,d\phi = R^3$$

For the third portion P_3, we have $\phi = \pi/2$, $d\phi = 0$, and

$$\mathbf{F} \cdot d\mathbf{l} = 0.$$

$$\int_{P_3} \mathbf{F} \cdot d\mathbf{l} = 0$$

In total we have the following result.

$$\int_P \mathbf{F} \cdot d\mathbf{l} = \frac{4}{3} R^3$$

Note that the result of a dotted line integral is a scalar.

C.5.2 Flux integrals

Example C.4. Find the flux integral for the vector field

$$\mathbf{F}(r, \theta, \phi) = r\hat{\mathbf{r}} + r \sin\theta \hat{\boldsymbol{\theta}} + r \sin\theta \cos\phi \hat{\boldsymbol{\phi}}$$

through the region enclosed by a triangle in the xy plane with vertices at $(x, y, z) = (0, 0, 0)$, $(x, y, z) = (2, 0, 0)$, and $(x, y, z) = (2, 2, 0)$.

Solution: The flux integral is

$$\int \mathbf{F} \cdot d\mathbf{a}$$

so the first order of business is to get an expression for $d\mathbf{a}$ in terms of the two variables we want to integrate over. The triangular region given suggests that we want to use Cartesian coordinates and integrate over x and y. The surface element $d\mathbf{a}$ is a vector, so we need an orientation for the surface. The orientation is always perpendicular to the surface, so it could be either $\hat{\mathbf{z}}$ or $-\hat{\mathbf{z}}$. An orientation was not given in the problem, so let's choose $\hat{\mathbf{z}}$ for our orientation.

$$d\mathbf{a} = dx\, dy \hat{\mathbf{z}}$$

Next, we want an expression for $\mathbf{F} \cdot d\mathbf{a}$ in terms of the two variables we want to integrate over. We notice that the vector field is given in spherical coordinates and the surface is described with Cartesian coordinates. When we take the dot product, we will have terms such as $\hat{\mathbf{r}} \cdot \hat{\mathbf{z}}$ showing up. Let's figure these out in advance. Starting with expressions for the spherical unit vectors in terms of the Cartesian unit vectors,

$$\begin{aligned}
\hat{\mathbf{r}} &= \frac{x\hat{\mathbf{x}} + y\hat{\mathbf{y}} + z\hat{\mathbf{z}}}{\sqrt{x^2+y^2+z^2}} = \sin\theta\cos\phi\hat{\mathbf{x}} + \sin\theta\sin\phi\hat{\mathbf{y}} + \cos\theta\hat{\mathbf{z}} \\
\hat{\boldsymbol{\theta}} &= \cos\theta\cos\phi\hat{\mathbf{x}} + \cos\theta\sin\phi\hat{\mathbf{y}} - \sin\theta\hat{\mathbf{z}} \\
\hat{\boldsymbol{\phi}} &= -\sin\phi\hat{\mathbf{x}} + \cos\phi\hat{\mathbf{y}}
\end{aligned}$$

we take the dot product of each of these equations with $\hat{\mathbf{z}}$ to find

$$\begin{aligned}
\hat{\mathbf{r}}\cdot\hat{\mathbf{z}} &= \cos\theta \\
\hat{\boldsymbol{\theta}}\cdot\hat{\mathbf{z}} &= -\sin\theta \\
\hat{\boldsymbol{\phi}}\cdot\hat{\mathbf{z}} &= 0.
\end{aligned}$$

Now we write an expression for $\mathbf{F}\cdot d\mathbf{a}$.

$$\begin{aligned}
\mathbf{F}\cdot d\mathbf{a} &= (r\hat{\mathbf{r}}\cdot\hat{\mathbf{z}} + r\sin\theta\hat{\boldsymbol{\theta}}\cdot\hat{\mathbf{z}} + r\sin\theta\cos\phi\hat{\boldsymbol{\phi}}\cdot\hat{\mathbf{z}})\,dx\,dy \\
&= (r\cos\theta - r\sin^2\theta)\,dx\,dy \\
&= \left(z - \frac{x^2+y^2}{\sqrt{x^2+y^2+z^2}}\right)dx\,dy \\
&= -\sqrt{x^2+y^2}\,dx\,dy
\end{aligned}$$

In the last step, we used the fact that $z=0$ for the surface we are using.

Next, we need limits for our integral.

$$\int \mathbf{F}\cdot d\mathbf{a} = -\int_0^2\int_0^x \sqrt{x^2+y^2}\,dy\,dx$$

The inner integral goes from the line $y=0$ to the line $y=x$. The outer integral goes from the point $x=0$ to the point $x=2$.

All that remains is to evaluate the double integral. The inner integral is

$$\int_0^x \sqrt{x^2+y^2}\,dy$$

This is an integral over y with x held constant. From Gradshteyn and Ryzhik, 5th edition, integral 2.271.3 (page 105) I find this integral.

$$\int \sqrt{a+x^2}\,dx = \frac{1}{2}x\sqrt{a+x^2} + \frac{1}{2}a\ln(x+\sqrt{a+x^2})$$

For a, we'll substitute x^2, and for x we'll substitute y.

$$\begin{aligned}
\int_0^x \sqrt{x^2+y^2}\,dy &= \left[\frac{1}{2}y\sqrt{x^2+y^2} + \frac{1}{2}x^2\ln(y+\sqrt{x^2+y^2})\right]_{y=0}^{y=x} \\
&= \left[\frac{1}{2}x\sqrt{x^2+x^2} + \frac{1}{2}x^2\ln(x+\sqrt{x^2+x^2})\right] - \left[\frac{1}{2}x^2\ln(\sqrt{x^2})\right] \\
&= \frac{\sqrt{2}}{2}x^2 + \frac{1}{2}x^2\ln(x+\sqrt{2}x) - \frac{1}{2}x^2\ln x \\
&= \frac{\sqrt{2}}{2}x^2 + \frac{1}{2}x^2\ln(1+\sqrt{2}) \\
&= \frac{\sqrt{2}+\ln(1+\sqrt{2})}{2}x^2
\end{aligned}$$

We used the fact that x is everywhere nonnegative on our surface so that $\sqrt{x^2} = x$.

$$\begin{aligned}
\int \mathbf{F}\cdot d\mathbf{a} &= -\int_0^2\int_0^x \sqrt{x^2+y^2}\,dy\,dx \\
&= -\int_0^2 \frac{\sqrt{2}+\ln(1+\sqrt{2})}{2}x^2\,dx \\
&= -\frac{8}{3}\frac{\sqrt{2}+\ln(1+\sqrt{2})}{2} \\
&= -\frac{4}{3}[\sqrt{2}+\ln(1+\sqrt{2})] \approx -3.06
\end{aligned}$$

The negative result makes sense because the vector field points downward (in the $-z$ direction) in the xy plane.

Appendix D

A Catalog of Fields, Paths, Surfaces, and Volumes

This chapter is a literate Haskell file. We give it the module name `Catalog` so that we can load these Haskell definitions elsewhere if we wish. We also import some types and functions from the `CoordinateSystems` module in Chapter 16.

```
{-# OPTIONS_GHC -Wall #-}

module Catalog where

import CoordinateSystems
    ( ScalarField
    , cartesianCoordinates
    )
```

D.1 Scalar Fields

D.1.1 Scalar fields expressed in Cartesian coordinates

D.1.1.1 $f(x, y, z) = x^2 y^3 z^4$

```
scalarField111 :: ScalarField
scalarField111 p = let (x,y,z) = cartesianCoordinates p
                   in x**2 * y**3 * z**4
```

D.1.1.2 Electric potential produced by a point charge q at the origin

$$V(x,y,z) = \frac{1}{4\pi\epsilon_0} \frac{q}{\sqrt{x^2+y^2+z^2}}$$

D.1.1.3 $f(x,y,z) = x^4 + y^4 + z^4$

D.1.1.4 Electric potential from a plane of charge with surface charge density σ on the xy plane

$$V(x,y,z) = -\frac{\sigma}{2\epsilon_0} |z|$$

D.1.1.5 $f(x,y,z) = x^2 + y^2 + z^2$

D.1.1.6 $f(x,y,z) = 4x^2y^3z^4$

D.1.2 Scalar fields expressed in cylindrical coordinates

D.1.2.1 $f(s,\phi,z) = s^2 z \cos\phi$

D.1.2.2 Electric potential produced by a point charge q at the origin

$$V(s,\phi,z) = \frac{1}{4\pi\epsilon_0} \frac{q}{\sqrt{s^2+z^2}}$$

D.1.2.3 $f(s,\phi,z) = \ln(s/a)$

where a is a constant with dimensions of length.

D.1.2.4 Magnitude of electric field from a long line charge with linear charge density λ along the z axis

$$E(s,\phi,z) = \frac{\lambda}{2\pi\epsilon_0 s}$$

D.1.3 Scalar fields expressed in spherical coordinates

D.1.3.1 $f(r,\theta,\phi) = 1/r^2$

D.1.3.2 Electric potential produced by a point charge q at the origin

$$V(r,\theta,\phi) = \frac{1}{4\pi\epsilon_0}\frac{q}{r}$$

D.1.3.3 $f(r,\theta,\phi) = \frac{1}{\sqrt{\pi a^3}}e^{-r/a}$

where a is a constant with dimensions of length.

D.1.3.4 $f(r,\theta,\phi) = r^2(3\cos^2\theta - 1)$

D.1.3.5 $f(r,\theta,\phi) = r^2\sin\theta\cos\phi$

D.1.4 Scalar fields expressed in a coordinate-independent fashion

D.1.4.1 Electric potential produced by a point charge q at the origin

$$V(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\frac{q}{|\mathbf{r}|}$$

D.1.4.2 $f(\mathbf{r}) = \mathbf{b}\cdot\mathbf{r}$

where $\mathbf{b}$ is a constant vector.

D.1.4.3 $f(\mathbf{r}) = \mathbf{r}\cdot\mathbf{r}$

D.1.4.4 Electric potential produced by an ideal dipole with dipole moment $\mathbf{p}$ at the origin

$$V(\mathbf{r}) = \frac{1}{4\pi\epsilon_0}\frac{\mathbf{p}\cdot\mathbf{r}}{|\mathbf{r}|^3}$$

D.2 Vector Fields

D.2.1 Vector fields expressed in Cartesian coordinates

D.2.1.1 $\mathbf{F}(x,y,z) = xy\hat{\mathbf{x}} + yz\hat{\mathbf{y}} + yz\hat{\mathbf{z}}$

D.2.1.2 Electric field from a plane of charge with surface charge density σ on the xy plane

$$\mathbf{E}(x,y,z) = \begin{cases} \frac{\sigma}{2\epsilon_0}\hat{\mathbf{z}} & , \quad z > 0 \\ -\frac{\sigma}{2\epsilon_0}\hat{\mathbf{z}} & , \quad z < 0 \end{cases}$$

D.2.1.3 Electric field produced by a point charge q at the origin

$$\mathbf{E}(x,y,z) = \frac{q}{4\pi\epsilon_0}\frac{x\hat{\mathbf{x}} + y\hat{\mathbf{y}} + z\hat{\mathbf{z}}}{(x^2+y^2+z^2)^{3/2}}$$

D.2.1.4 $\mathbf{F}(x,y,z) = x^2\hat{\mathbf{x}} + y^2\hat{\mathbf{y}} + z^2\hat{\mathbf{z}}$

D.2.1.5 $\mathbf{F}(x,y,z) = xy^2\hat{\mathbf{x}} + xy^2\hat{\mathbf{y}} + 3a^3\hat{\mathbf{z}}$

where a is a constant with dimensions of length.

D.2.1.6 Electric field produced inside a planar slab by a uniform volume charge density ρ_0

$$\mathbf{E}(x,y,z) = \frac{\rho_0}{\epsilon_0}z\hat{\mathbf{z}}$$

D.2.1.7 Electric field from a plane of charge with surface charge density σ_0 on the yz plane

$$\mathbf{E}(x,y,z) = \begin{cases} \frac{\sigma_0}{2\epsilon_0}\hat{\mathbf{x}} & , \quad x > 0 \\ -\frac{\sigma_0}{2\epsilon_0}\hat{\mathbf{x}} & , \quad x < 0 \end{cases}$$

D.2.2 Vector fields expressed in cylindrical coordinates

D.2.2.1 Electric field from a long line charge with linear charge density λ along the z axis

$$\mathbf{E}(s,\phi,z) = \frac{\lambda}{2\pi\epsilon_0 s}\hat{\mathbf{s}}$$

D.2.2.2 Electric field from a plane of charge with surface charge density σ on the $z=0$ plane

$$\mathbf{E}(s,\phi,z) = \begin{cases} \frac{\sigma}{2\epsilon_0}\hat{\mathbf{z}} & , \quad z>0 \\ -\frac{\sigma}{2\epsilon_0}\hat{\mathbf{z}} & , \quad z<0 \end{cases}$$

D.2.2.3 $\mathbf{F}(s,\phi,z) = s^2\hat{\mathbf{s}}$

D.2.2.4 $\mathbf{F}(s,\phi,z) = s^2\hat{\boldsymbol{\phi}}$

D.2.2.5 $\mathbf{F}(s,\phi,z) = sz\hat{\mathbf{s}} + s^2\sin\phi\hat{\mathbf{z}}$

D.2.2.6 $\mathbf{F}(s,\phi,z) = s^2\cos\phi\hat{\mathbf{s}} + s^2\sin\phi\hat{\boldsymbol{\phi}}$

D.2.2.7 Electric field produced inside a long cylinder by a uniform volume charge density ρ_0

$$\mathbf{E}(s,\phi,z) = \frac{\rho_0}{2\epsilon_0}s\hat{\mathbf{s}}$$

D.2.3 Vector fields expressed in spherical coordinates

D.2.3.1 Electric field produced by a point charge q at the origin

$$\mathbf{E}(r,\theta,\phi) = \frac{1}{4\pi\epsilon_0}\frac{q}{r^2}\hat{\mathbf{r}}$$

D.2.3.2 Electric field produced by an ideal dipole at the origin with dipole moment p in the z direction

$$\mathbf{E}(r,\theta,\phi) = \frac{p}{4\pi\epsilon_0 r^3}(2\cos\theta\hat{\mathbf{r}} + \sin\theta\hat{\boldsymbol{\theta}})$$

D.2.3.3 $\mathbf{F}(r,\theta,\phi) = r\hat{\mathbf{r}} + r\sin\theta\hat{\boldsymbol{\theta}} + r\sin\theta\cos\phi\hat{\boldsymbol{\phi}}$

D.2.3.4 $\mathbf{F}(r,\theta,\phi) = r^2\hat{\mathbf{r}} + r^2\sin\theta\cos\theta\hat{\boldsymbol{\theta}} + r^2\sin^2\theta\cos^2\phi\hat{\boldsymbol{\phi}}$

D.2.3.5 Electric field produced inside a sphere by a uniform volume charge density ρ_0

$$\mathbf{E}(r,\theta,\phi) = \frac{\rho_0}{3\epsilon_0}r\hat{\mathbf{r}}$$

D.2.3.6 $\mathbf{F}(r,\theta,\phi) = r\sin\theta\hat{\boldsymbol{\theta}}$

D.2.3.7 $\mathbf{F}(r,\theta,\phi) = r\sin\theta\hat{\boldsymbol{\phi}}$

D.2.4 Vector fields expressed in a coordinate-independent fashion

D.2.4.1 Electric field produced by a point charge q at the origin

$$\mathbf{E}(\mathbf{r}) = \frac{q}{4\pi\epsilon_0}\frac{\mathbf{r}}{|\mathbf{r}|^3}$$

D.2.4.2 Electric field produced by a point charge q_1 at position $\mathbf{r}_1$

$$\mathbf{E}(\mathbf{r}) = \frac{q_1}{4\pi\epsilon_0}\frac{\mathbf{r}-\mathbf{r}_1}{|\mathbf{r}-\mathbf{r}_1|^3}$$

D.2.4.3 $\mathbf{F}(\mathbf{r}) = 3\mathbf{r}$

D.2.4.4 $\mathbf{F}(\mathbf{r}) = (\mathbf{r}\cdot\mathbf{r})\mathbf{r}$

D.3 Paths

D.3.1 Paths expressed in Cartesian coordinates

D.3.1.1 The path from $(0,0,0)$ to $(x_0,0,0)$ along the x axis, then to $(x_0,y_0,0)$ along a straight line parallel to the y axis, then to (x_0,y_0,z_0) along a straight line parallel to the z axis

D.3.1.2 The path from $(0,0,0)$ to $(0,0,z_0)$ along the z axis, then to $(0,y_0,z_0)$ along a straight line parallel to the y axis, then to (x_0,y_0,z_0) along a straight line parallel to the x axis

D.3.1.3 The straight-line path from $(0,0,0)$ to (a,a,a)

where a is a constant with dimensions of length.

D.3.1.4 The path around a square from $(0,0,0)$ to $(a,0,0)$ to $(a,a,0)$ to $(0,a,0)$ to $(0,0,0)$

where a is a constant with dimensions of length.

D.3.1.5 The boundary of surface D.4.1.2

D.3.1.6 The straight-line path from $(0, 0, 3)$ to $(2, 4, 3)$

D.3.1.7 The path from $(1, 2, 3)$ to $(4, 5, 3)$ that runs from $(1, 2, 3)$ to $(4, 2, 3)$ along a straight line and then from $(4, 2, 3)$ to $(4, 5, 3)$ along a straight line

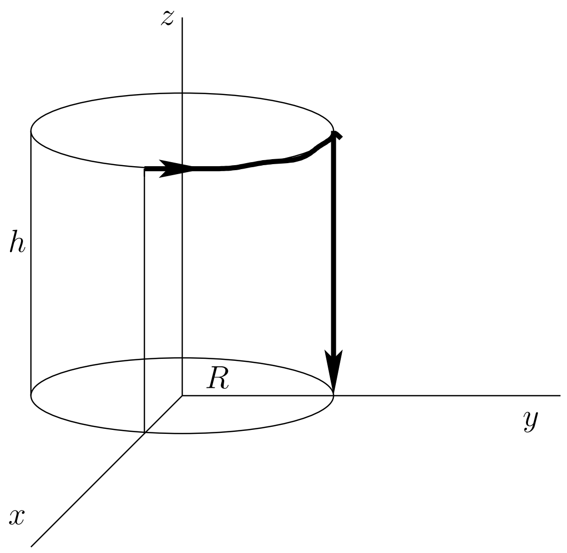
D.3.2 Paths expressed in cylindrical coordinates

D.3.2.1 The circular path with radius R in the $z = 0$ plane starting at $\phi = 0$ and going to $\phi = 2\pi$

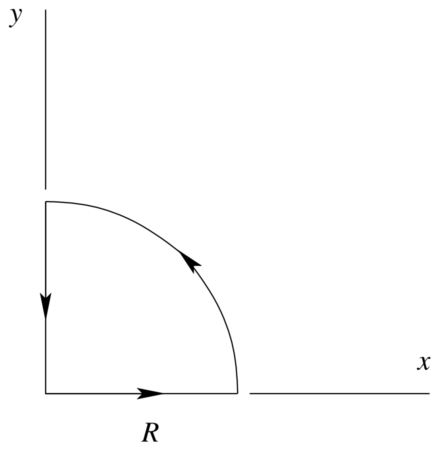
D.3.2.2 The straight path in the $z = 0$ plane from the origin along $\phi = \pi/4$ until $s = R$

D.3.2.3 The boundary of surface D.4.2.1

D.3.2.4 the path on the surface of a cylinder of radius R that goes (i) along a circular arc from a point at $(x, y, z) = (R, 0, h)$ to a point at $(x, y, z) = (0, R, h)$, and then (ii) along a straight-line path from $(x, y, z) = (0, R, h)$ to $(x, y, z) = (0, R, 0)$



D.3.2.5 The closed path shown in the figure



D.3.3 Paths expressed in spherical coordinates

D.3.3.1 The path on the surface of a sphere of radius R that goes (i) from the north pole at $(x, y, z) = (0, 0, R)$ to the equator at $(x, y, z) = (R, 0, 0)$, and then (ii) along the equator to the point $(x, y, z) = (R/\sqrt{2}, R/\sqrt{2}, 0)$

D.3.3.2 The straight-line path from the origin to the point with spherical coordinates $(r, \theta, \phi) = (2, \pi/6, \pi/4)$.

D.3.3.3 The path from $(r, \theta, \phi) = (R, \pi/4, 0)$ to $(r, \theta, \phi) = (R, \pi/4, \pi/2)$ along which $r = R$ and $\theta = \pi/4$.

D.3.3.4 The path from $(r, \theta, \phi) = (R, 0, 2\pi/3)$ to $(r, \theta, \phi) = (R, \pi/2, 2\pi/3)$ along which $r = R$ and $\phi = 2\pi/3$.

D.4 Surfaces

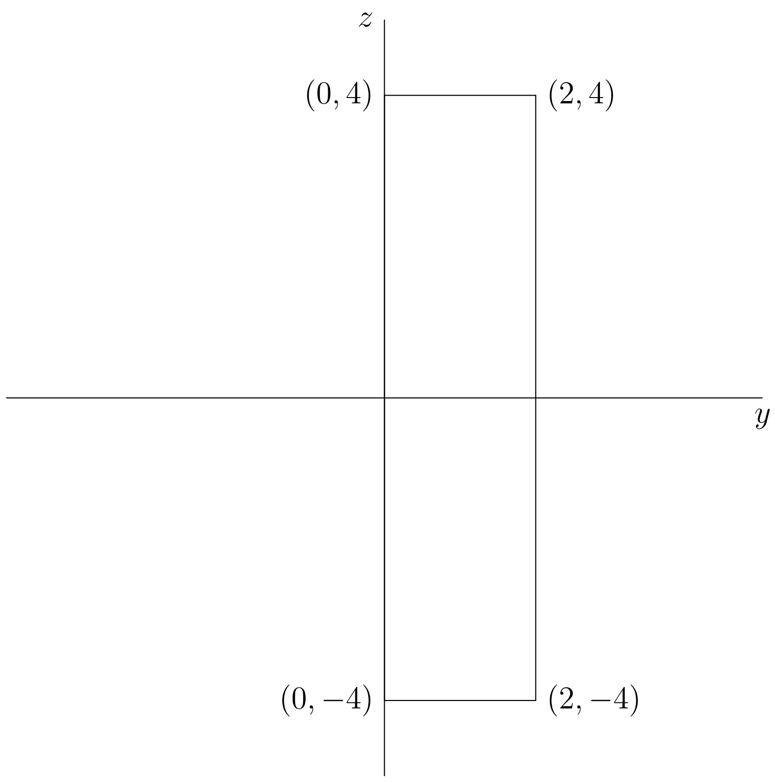
D.4.1 Surfaces expressed in Cartesian coordinates

D.4.1.1 The boundary of volume D.5.1.1

D.4.1.2 The region enclosed by a square in the xy plane with side length L centered at the origin

If an orientation is needed, use $\hat{\mathbf{z}}$.

D.4.1.3 The rectangular region shown in the figure



If an orientation is needed, use $\hat{\mathbf{x}}$.

D.4.1.4 The region enclosed by a triangle in the xy plane with vertices at $(x, y, z) = (0, 0, 0)$, $(x, y, z) = (2, 0, 0)$, and $(x, y, z) = (2, 2, 0)$

D.4.1.5 The rectangle with vertices $(x, y, z) = (a, b, c)$, $(x, y, z) = (-a, b, c)$, $(x, y, z) = (-a, -b, c)$, and $(x, y, z) = (a, -b, c)$

The constants a, b, and c are all positive. If an orientation is needed, use $\hat{\mathbf{z}}$.

D.4.1.6 The rectangle with vertices $(x, y, z) = (a, b, c)$, $(x, y, z) = (a, -b, c)$, $(x, y, z) = (a, -b, -c)$, and $(x, y, z) = (a, b, -c)$

The constants a, b, and c are all positive. If an orientation is needed, use $\hat{\mathbf{x}}$.

D.4.2 Surfaces expressed in cylindrical coordinates

D.4.2.1 A disk in the $z = 0$ plane with radius R, centered at the origin

If an orientation is needed, use $\hat{\mathbf{z}}$.

D.4.2.2 The boundary of volume D.5.2.2

D.4.3 Surfaces expressed in spherical coordinates

D.4.3.1 A sphere of radius R centered at the origin

D.5 Volumes

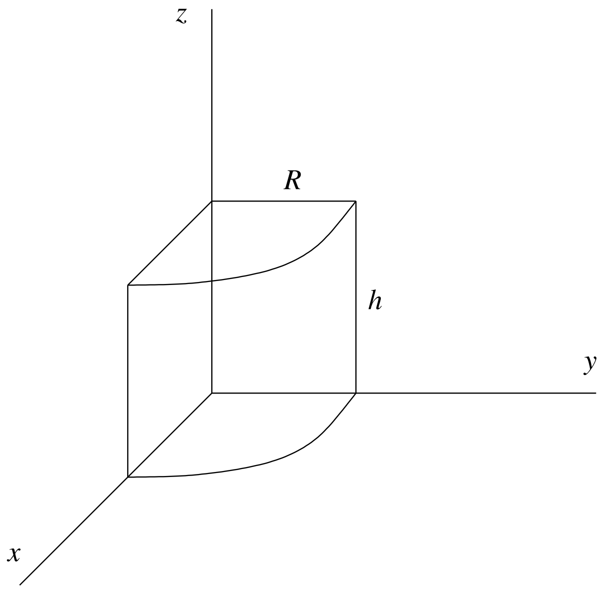
D.5.1 Volumes expressed in Cartesian coordinates

D.5.1.1 A cube with side length L centered at the origin

D.5.1.2 The rectangular box in which $0 \leq x \leq a$, $0 \leq y \leq b$, and $0 \leq z \leq c$, where a, b, and c are some constants with dimensions of length

D.5.2 Volumes expressed in cylindrical coordinates

D.5.2.1 The quarter cylinder with height h and radius R shown in the diagram below



D.5.2.2 A cylinder with height h and radius R centered at the origin, with the cylinder axis on the z axis.

D.5.3 Volumes expressed in spherical coordinates

D.5.3.1 A ball of radius R centered at the origin

D.5.3.2 The upper half of a ball of radius 2, with the center of the ball at the origin

Index