The Calculus of Programming

John C. Lusth

February 17, 2010
13 Breaking Up is Hard to Do 75
   13.1 Partial Fractions ................................................. 75
   13.2 Inverse Functions .................................................. 76

14 The Rich get Richer 79

15 Auld Lang Sine 81
   15.1 The general sine wave ........................................... 82
   15.2 Implementing the general sine wave ............................ 83
   15.3 Differentiating sine and cosine ................................. 84
   15.4 What is -sin? ......................................................... 86
   15.5 Questions ............................................................. 86

16 Do What You Can 87

17 And Now For Something Completely Different 89
   17.1 Summing a series .................................................. 89
       17.1.1 The recursive way ........................................... 90
       17.1.2 An iterative approach ...................................... 92
   17.2 Summing as an abstraction .................................... 97
       17.2.1 Pseudo-infinite sequences ................................. 98
       17.2.2 Streams, the easy way ..................................... 101
   17.3 Numeric Integration .............................................. 102
   17.4 The power of streams .......................................... 106
   17.5 Questions ............................................................. 107
Chapter 1

Terrifying Concepts

Probably the two most terrifying concepts in the freshman world are Calculus and Programming. They certainly were for me. I was a C student when it came to Calculus and an B student when it came to programming. Let me assure you that both concepts are quite simple, notwithstanding my grades as a freshman. Given that both calculus and programming are so simple, my freshman grades are most embarrassing.

Let’s start with calculus. I don’t know how or why I got it into my head that calculus was a difficult subject. As pointed out by Sylvanus P. Thompson (herein SPT) in the analogous chapter of Calculus Made Easy (herein CME), you simply have to understand the two symbols you will encounter over and over again when discussing Calculus. They are $f$ and $d$. The first symbol, $\int$ (the integral symbol), simply means ‘to sum up’. The second symbol, $d$ (the differentiation symbol), means simply ‘a tiny piece of’. Placed together, as in

$$\int x^2 \, dx$$

they simply means to sum up all the tiny pieces of the curve $x^2$.

Let me add that the purpose of differentiation is just to find the slope of a curve and that the purpose of integration is to find the area under a curve. That’s all.

With regards to programming, it also is rather simple. To write a program, one simply calls functions to do the work you need to do. Sometimes the function you need to call already exists. Sometimes it doesn’t and you have to build it. But guess what? You are going to build the function from other functions, some of which already exist and some that do not. Simple, right?

I’ve used the word simple and its variants an awful lot, but I do so because it really is all so simple.
Chapter 2

Let’s Get Small

Recall from the last chapter that the symbol \( d \) with respect to Calculus means ‘to take tiny pieces of’. Let’s practice taking small pieces of a number and, at the same time, learn some programming. And, because you must, as future Computer Scientists, you must learn to let go and be wild, we will take a small piece of a small piece, and a small piece of that very small piece, and so on and so on, \textit{ad infinitum}.

At this point, you should install Sway (see http://sway.cs.ua.edu) on your system.

2.1 Using the Sway Interpreter

We begin by starting the Sway interpreter, Sway being the Programming Language you will learn. Once started, the interpreter rewards you with a prompt:

```
sway>
```

This prompt signals to you to enter some code. When you do so, the interpreter will tell you the result of running (or evaluating or executing) that code.

```
sway> 3;
INTEGER: 3
```

To exit the interpreter, type \(<\text{Ctrl}>-d\) or a \(<\text{Ctrl}>-c\). These key strokes are generated by holding the Contol key down while at the same time tapping the ‘d’ or ‘c’ key.\cite{1} Here, we have entered a 3 followed by a semicolon (the semicolon tells the interpreter that we are done entering code). Sway (actually the Sway interpreter) responds by saying 3 is an integer with a value of 3.

For more on Sway primitives, see \textit{The Sway Reference Manual (TSRM)}, Chapter 2.

Of course, we already knew that 3 is an integer, so the interpreter doesn’t seem to be all that useful. But did you know that 43 times 112 is 4816?

```
sway> 43 * 112;
INTEGER: 4816
```

\footnote{If you restart the Sway interpreter, you can recover your previous commands by using the up arrow on your keyboard. If you go too far up, you can use the down arrow to move through the list of previous commands in the opposite direction. You can edit the previous commands as well.}
Sway does! Let’s use the fact that the interpreter is pretty good at math to figure out what a small part of a big number is. Let’s assume that a small part is \( \frac{1}{16} \) of the whole.² First, let’s figure out what \( \frac{1}{16} \) is as a decimal number, or real number, since that will be easier to type in:

```
sway> 1/16;
SOURCE CODE ERROR
file stdin,line 1
an expression was expected, found token of type BAD_NUMBER
error occurred prior to: 16;[END OF LINE]
```

Uh oh. Sway did not like the ‘1’ followed so closely by the ‘/’. In fact, Sway requires spaces around things like division signs in most instances. Let’s try again:

```
sway> 1 / 16;
INTEGER: 0
```

Better, at least we got an answer instead of an error. However, it doesn’t seem that the interpreter is that good at math after all. If we put on our Sherlock Holmes cap and ponder, we see that the interpreter said the result of dividing 1 by 16 is an integer, but we know it should be a real number. It turns out that the Sway language, like most programming languages, uses a rule that combining two things of the same type yields a result of the same type. In this case, zero happens to be the largest integer that is less than the desired result. In other words, the interpreter truncated the fractional part of the real number and gave us the integer that was left. Let’s experiment and see if this is so:

```
sway> 7 / 2;
INTEGER: 3
```

Seems to be. So getting back to our original problem, how do we find out what is 1 divided by 16 as a real number? Let’s enter the numbers as real numbers instead of integers:

```
sway> 1.0 / 16.0;
REAL: 0.0625000000
```

That’s much better!

Now let’s figure out what a small part of a million is (using our assumption of what is small):

```
sway> 1000000 * 0.0625;
REAL: 62500.0000000000
```

Sixty-two and a half thousand. Still big in an absolute sense, but much smaller that a million. Being wild and crazy, let’s take a small part of a small part:

```
sway> 1000000 * .0625 * .0625;
REAL: 3906.2500000000
```

About 4000. Much smaller.

²Computer Scientists love numbers that are a power of 2, as in 1, 2, 4, 8, 16, 32, and so on. The inverses of those numbers are much loved, as well.
2.2 Using Variables

Shall we continue taking ever smaller parts?

Before we do, I must admit that I am, at heart, a lazy person, as are most Computer Scientists. Typing in all those numbers is just too much work! I am going to use two short symbols to represent both the million and the fraction:

```sway
var x = 1000000;
INTEGER: 1000000
var f = .0625;
REAL: 0.0625000000
```

What I’ve done is created a variable to stand in for the million and a variable to stand in for the fraction, $x$ and $f$ respectively. Now I can use those variables instead of the numbers. Let’s check to see if I did things right:

```sway
x * f * f;
REAL: 3906.2500000000
```

Looks like I did. Let’s go a step further:

```sway
x * f * f * f;
REAL: 244.1406250000
```

It seems variables are a nice way to reduce the amount of typing needed. The only drawback is remembering what a variable stands for. This is why it is so important to name your variables in such a way as to make it easy for you to recall their meanings. Generally, single letter variable names are not a good idea (although there are exceptions to this rule).

To learn more, see TSRM, Chapter 5: Variables.

2.3 Using functions

We could go further with this, but you have yet to understand the depths of my laziness. Even typing in $f$ repeatedly is too much for my sensibilities. I am going to define (or write) a function named `smaller` to do the work for me (you can cut and paste this function into the interpreter if you are lazy like me and don’t want to type it in yourself):

```sway
function smaller(amount,fraction)
{
    inspect(amount * fraction);
}
```

Whether you paste it in or type it in, you should get the following response from the interpreter:

---

3Lazy, as in I abhor doing work that could (and should) be automated.

4A variable can be thought of as a name for something else. However, it is not the something else, just as your name is not you, but a handy way for people to refer to you.
CHAPTER 2. LET’S GET SMALL

sway> function smaller(amount,fraction)
more> {
more> inspect(amount * fraction);
more> }
FUNCTION: <function smaller(amount,fraction)>

The more prompt indicates that the Sway interpreter is expecting some more code.

A function abstracts a series of actions much like a variable abstracts a value. There are two things you must do when dealing with functions, (1) define them and (2) call them. Here, we have just defined a function; now we need to call it. We will call it by typing the name of the function followed by a parenthesized list of arguments. Sometimes we say this is “passing the arguments to the function”.

When calling the function smaller, the values of the arguments will be bound to the variables amount and fraction, which are found after the function name in its definition. These variables are known as the formal parameters of the function. This passing and binding, in essence, defines those variables for us. Note that the arguments and formal parameters do need to be separated by whitespace; the comma serves as a separator.

After these implicit variable definitions, the code between the curly braces will be evaluated as if you had typed them in directly to the interpreter. This is why I said in the previous chapter that a function does the work for you. If we look at the code (or body) of the function we just defined, we see that a function named inspect is being called. Since we haven’t defined inspect, we can assume that this function already exists within the interpreter. By its name, we can guess that it will tell us something about what happens when we multiply amount by fraction:

sway> smaller(x,f);
amount * fraction is 62500.00000
REAL: 62500.00000

There is a whole lot going on here that needs explanation. The first is that the value of x (1000000) was bound to the formal parameter amount of the function smaller. Likewise, the value of f (0.0625) was bound to the formal parameter fraction. Then the body of the function smaller was evaluated, triggering a call to the inspect function. What inspect does is print out its literal argument, followed by the string " is ", followed by the value of its argument. Since the call to inspect is that last thing smaller does, whatever inspect returns is returned by smaller. This return value appears to be the evaluated argument.

Note that interpreter reported two things. The first is the string produced by inspect. The second is the report of the return value as we have seen before. We call an action that has an effect outside the called function (in this case, the printing by inspect) a side effect.

To learn more, see TSRM, Chapter 6: Functions.

2.4 Recursion

Now, at this point, you are probably thinking that, not only am I lazy, I must be a rather dim bulb as well, because I spent a lot more effort writing the smaller function and calling it than I would have simply typing:

sway> x * f;
REAL: 62500.00000

\footnote{The variable smaller is not really a function, but a handy name for the function we defined. But rather than say ‘the function named by the variable smaller’, we often say ‘the function smaller. Technically, that is incorrect, but is simpler to say.}
If that was all I was going to do, you would be quite right in your assessment. But I am not finished yet. Now I am going to make smaller much, much more powerful:

``` sway
function smaller(amount,fraction)
{
    inspect(amount * fraction);
    smaller(amount * fraction,fraction);
}
```

Notice that I have added a call to the smaller function after the call to the inspect function. When a function calls itself, it is said to be a recursive function, that it exhibits the property of recursion and that at the point of the recursive call, the function recurs. Notice further that the first argument to smaller in that internal call will send a (hopefully) smaller number to the smaller function. In that call, smaller will be called again with yet an even smaller number, and so on. Let’s try it:

``` sway
sway> smaller(1000000,.0625);
amount * fraction is 62500.0000000000
amount * fraction is 3906.2500000000
amount * fraction is 244.1406250000
amount * fraction is 15.2587890625
...
amount * fraction is 0.0000000000e+00
amount * fraction is 0.0000000000e+00
amount * fraction is 0.0000000000e+00
encountered a fatal error...
stack overflow.
```

Wow! Unless you have very quick eyes, all you saw is the bottom part of this output. What happened? What we did was to define a function that fell into an infinite loop when we called it. An infinite loop occurs because we never told our function when to stop calling itself. Thus, it tried to call itself ad infinitum. Of course, a computer has a limited amount of memory, so in this particular case, the calls could not go on forever. Let’s redefine our function so that it pauses after every inspection so we can slow down the output:

``` sway
function smaller(amount,fraction)
{
    inspect(amount * fraction);
    pause();
    smaller(amount * fraction,fraction);
}
```

You can start up the Sway interpreter again and paste in our revised function definition. Now when we call our function we see this output (assuming you repeatedly hit the enter key on your keyboard):

``` sway
sway> smaller(1000000,.0625);
```

---

6 Some poor souls will use the verb recurse instead of the proper recur. Recur means to call oneself, recurse means to swear again. Do not make this mistake, otherwise people will deem you ignorant.

7 There are clever languages in which some infinite recursive loops do not use up computer memory. Someday, Sway will be that clever and this section will need to be rewritten.
amount * fraction is 62500.0000000000
amount * fraction is 3906.2500000000
amount * fraction is 244.1406250000
amount * fraction is 15.2587890625
amount * fraction is 0.9536743164
amount * fraction is 0.0596046448
amount * fraction is 0.0037252903
amount * fraction is 0.0002328306

You can stop the interpreter by typing in a <Ctrl>-c which is entered from your keyboard by holding the Control key down while tapping the ‘c’ once.

We can see that amount * fraction gets very small very quickly. If you start again and keep going, you will see that amount * fraction eventually reaches zero. Theoretically, it doesn’t, but at some point the quantity gets smaller that can be represented by Sway, so Sway reports zero.

You should read more about TSRM, Chapter 13: Recursion and learn about TSRM, Chapter 11: If expressions before continuing.

Let’s try a new version of the function, this one calls itself a given number of times before stopping:

```sway
function smaller(amount,fraction,count)
{
    if (count == 0)
    {
        :done;
    }
    else
    {
        inspect(amount);
        smaller(amount * fraction,fraction,count - 1);
    }
}
```

This time, we have a new formal parameter count, which represents the number of times to the function recursively calls itself again. We’ve also changed the call to inspect to print out the value of amount. Note that the recursive call not only makes amount smaller, it makes count smaller as well. When count gets small enough, the function returns the symbol done.

We must remember to add an extra argument to the call. Let’s send in 8 as the number of times to recursively call:

```
sway> smaller(1000000,.0625,8);
amount is 1000000
amount is 62500.000000
amount is 3906.2500000
amount is 244.14062500
```
Yay! Our program stopped recurring infinitely. Formally, the if inside the function has two cases: the base case, which does not contain a recursive call and a recursive case, which does. Infinite recursive loops occur when the base case is never reached, usually do to some misstatement of the base case condition or a misunderstanding of the range of values taken on by the formal parameter being tested in the base case condition.

2.5 Back to Calculus

What did this exercise with finding ever smaller numbers have to do with calculus? Well, the \( d \) symbol means ‘take a tiny piece of’. How tiny? Infinitesimally small, or in other words, the value computed by an infinite number of recursive calls to smaller. Of course, this is smaller than we can fathom, but that is not a big deal. We can’t fathom how our brains work, but we get along OK, or at least most of us do.

2.6 Questions

All formulas are written using Grammar School Precedence.

1. What happens when you combine an integer and a real number?

2. This question and subsequent questions refer to the last version of smaller. Why did the call to smaller return a real number?

3. Redefine smaller so that it prints out nine inspections rather than eight, for the same initial value of 8.

4. What happens when zero is passed as the count to smaller?

5. What happens when -1 is passed as the count to smaller? Why?

6. What happens if the recursive call in smaller is replaced by smaller(x * f,f,n)? Why?

7. Write a Sway function to represent \( y = 2x^a \), and evaluate it for \( x = 1, 2, 3 \), and \( a = 2, 4, 6 \).

8. Using your Sway function for \( y \) from the previous problem, write a new Sway function \( z = ay \) and evaluate as before.

9. A series is a sequence of terms that are added together. If, as more and more terms are added together, the sum of the terms get closer and closer to a number \( k \), then \( k \) is said to be the limit of that series. CME explains this by referring to Zeno’s paradox. Let’s say you start a distance 1 away from a wall and with each step, you travel half the remaining distance to the wall (why is this a paradox?). Using recursion and Sway, define a function zeno(n) that demonstrates this process. The argument to zeno is the number of steps taken and the return value should be the total distance travelled. What is the limit of the zeno series?

10. Let \( g(x) = x^2 \) and \( h(x) = (g(1 + x) - g(1 - x))/(3x) \). Calculate \( h(1), h(2), h(3) \), and \( h(0.5) \). Express \( h(x) \) in terms of \( x \). Draw a graph of \( h \) in the range \(-2 \leq x \leq 2\).

11. The formula for the chance of an NPC (Non-Player Character) landing a crushing blow in the World of Warcraft (a popular Massively Multiplayer Online Role-Playing Game or MMORPG), is \( (NPCLevel * 5 - PlayerDefense) * 2\% - 15\% \). For a level 70 NPC, express this as a function of PlayerDefense(x) and give the value of defense that makes the chance 0.
12. The amount of experience to level a character is \( h(x) = g(x) \times (1344 - ((69 - x) \times ((69 - x) \times 4 + 3))) + 155 \)
where \( x \) is the character level and where \( g(x) = 5x + 45 \) is the basic amount of experience earned for dispatching a mob of level equal to the character. Calculate \( h(61) \), \( h(69) \), \( h(0.5) \), and \( h(1) \). Express \( h(x) \) in terms of \( x \). Plot a graph of \( h \) for \( 61 \leq x \leq 69 \).
Chapter 3

A Little Bit of This, A Little Bit of That

In the last chapter, you were introduced to the idea of a variable. You can think of a variable as a spot in the computer’s memory where you can store a value:

```
sway> var x = 1000000;
INTEGER: 1000000
sway> x;
INTEGER: 1000000
```

Here I’ve named that spot $x$ and I’ve stored the number 1000000 at that spot. Once I’ve reserved this spot, I can change the value there by assigning a new value to the variable. Here’s an example:

```
sway> x = 42;
INTEGER: 42
sway> x;
INTEGER: 42
```

The equals sign is used to indicate that the value assigned to a variable is to be changed. Notice that I did not use the word `var` this time. When I use `var`, it means reserve me a new spot. When I don’t use `var`, it means update a previously reserved spot.

To learn more about assignment, please see *The Sway Reference Manual*, Chapter 7.

What happens if we try to update a spot without reserving it first?

```
sway> y = 3;
EVALUATION ERROR: :assignError
stdin,line 3: assignment: variable y is undefined
```

We get a message that $y$ is undefined. This is the interpreter’s way of telling us that we forgot to reserve a spot for $y$ using `var`. Sway knows we didn’t define the variable because it looked for it in the current environment and, failing, in the context environment and, failing, in the context of the context, and so, until it ran out of places to look.

The word variable implies that the value can vary, which is exactly what we did with $x$; we changed its value. In some programming languages, there are analogous beasties named constants, only once created, you cannot vary them. In Sway, there are no constants, per se. Should we need a constant, we will use a
variable instead, but we will make a pledge not to change a variable’s value. To remind us that we should not change its value, we will use capital letters (mostly) in naming the variable:

```
sway> var PI = 3.14159;
REAL: 3.1415900000
```

Making this pledge is known as following a convention. Note that variable names can be (and generally should be) more than a single letter. Note also that SPT in Chapter 3 of CME uses lower case letters early in the alphabet to indicate constants and lower case letters late in the alphabet to indicate variables. That is a different convention.

### 3.1 Adding a little bit

Here’s a problem that is a little bit simpler than the one from CME\(^1\):

Consider a rectangle whose width and height are related. Suppose we increase the width a little bit. What happens to the height, assuming the new height is still one half of the new width? Since the height is calculated from the width, we say that width is the independent variable and that height is the dependent variable. First, let’s define a function that gives us the height, \( h \), given the width, \( w \):

```
function h(w)
{
    w / 2.0;
}
```

Note that when we write functions like this, it is almost always the case that the dependent variable is the name of the function and the independent variable is the name of the formal parameter. Use this rule to help you define the functions you need.

We can test out our new function:

```
sway> h(100);
REAL: 50.0000000
```

This confirms our suspicion that if the width is 100 units, the height must be 50 units.

Calculus is often concerned with the change in the dependent variable given a change in the independent variable, particularly the ratio of the two changes.

Now, let’s figure out that ratio for the above rectangle, using a small change in width:

\(^1\)This figure is drawn using Cheesy ASCII (tm) Graphics (for fast loading).
The change in the height is simply the new height minus the old height. Now we can compute the ratio:

\[
\text{ratio(w)} = \frac{\text{dh}}{\text{dw}}
\]

Another thing Calculus is concerned with is does this ratio change with different values for the independent variable? The lazy person’s way to answer these questions is, you guessed it, to write a function to do the work.

```javascript
function ratio(w)
{
    var dw = 0.01;
    var dh = h(w + dw) - h(w);
    dh / dw;
}
```

Note that we pass in the value of \( w \) and then compute the ratio at that particular width. This allows us to try a wide variety of values quite easily:

```
sway> ratio(40);
REAL: 0.5000000000
sway> ratio(20);
REAL: 0.5000000000
```

This ratio function is quite useful! We see that the ratio of \( \frac{dy}{dx} \) does not seem to ever change. But suppose we now start looking at rectangles with a different relationship between width and height. All of a sudden, our ratio function is looking a lot less useful. This is because we hard-wired the height function. We can fix this by passing in the height function as an additional argument. Now we can easily try out new height functions. By doing so, we have made the ratio computing function more general.

```javascript
function ratio(h,w)
{
    var dw = 0.01;
    var dh = h(w + dw) - h(w);
    dh / dw;
}
```

It should work just the same:

```
sway> ratio(h,40);
REAL: 0.5000000000
sway> ratio(h,20);
REAL: 0.5000000000
```

---

2 We hard-wire something when we treat it as a constant, something that doesn’t change.
Let’s try a different height function:

\[
\text{function } g(w) \text{ }
\{ \\
\quad w \times w; \\
\}
\]

With this height function, the height \( g \), grows as the square of the width. What does \( \text{ratio} \) produce now? Let’s try one width and see if it matches what we expect from the power rule:

\[
\text{sway> ratio}(g,100); \\
\text{REAL: 200.01000000}
\]

The power rule tells us that the derivative of:

\[
h = w^2
\]

should be:

\[
\frac{dh}{dw} = 2w
\]

Therefore, we would expect that for a width of 100, the ratio function should return 200. It almost does, but not quite. Why?

The reason is the ratio function would return exactly 200 if the value we chose for \( dw \) was infinitely small. Unfortunately, we cannot represent an infinitely small number numerically in a physical computer, so our result incorporates some error. However, the smaller the value we choose for \( dw \), the smaller the error. Try this: edit the ratio function and change the value of \( dw \) to 0.00001 and rerun the call to ratio. We get:

\[
\text{sway> ratio}(g,100); \\
\text{REAL: 200.00001005}
\]

With the smaller value for \( dw \), we confirm a smaller error in the ratio.

Now let’s try a different width:

\[
\text{sway> ratio}(g,200); \\
\text{REAL: 400.00001027}
\]

For this new width, 200, the exact ratio appears to be 400, exactly what is predicted by the power rule. \(^3\)

For the height function \( g \), we see that different starting widths yield different ratios. A good deal of calculus deals with explaining why such ratios change. Also, this process of computing ratios is known as the \textit{taking the derivative numerically}.

\(^3\)You can trust integers, because they are exact. But you should never trust real numbers completely, because they are approximations. Sway real numbers are only valid to 15 significant digits.
3.2 Questions

1. All formulas are written using Grammar School Precedence.

2. Does it matter what variable name we use as a formal parameter when defining a function?

3. Draw a picture of a labelled rectangle so that the height is the independent variable, keeping the relationship that the height is one half the width.

4. In the function ratio, \(dh\) is computed from \(h(w + dw) - h(w)\). If we assume that \((w + dh)\) is equivalent to \((w) + h(dw)\) then \(dh\) is \((w) + h(dw) - h(w)\) or simply \((dw)\). Define a new function named \(ratio2\) to test whether this assumption is valid.

5. Which function, \(ratio1\) or \(ratio2\), is better for very small changes? Why?

6. Consider a rectangle whose area never varies. That is, if the width is increased, the height decreases to compensate so that the area of the rectangle remains the same. Define a function that computes the height of this rectangle given the width and the area.

7. Do we observe the same behavior in the ratios as we did with function \(g\)?

8. “Change is constant”, but does not have to be a constant. Most of us are comfortable with the notion of a constant rate of change. For instance the change in height versus distance as described by a straight line, for instance \(y = 5x + 4\). Use the ratio function to explore how \(y\) changes as \(x\) changes. Produce a table of \(y\), \(x\), and \(dy/dx\).

9. The previous problem had a constant rate of change. Often, however, the rate of change of the dependent variable changes with the independent variable. Let’s take as an example the function for area of a square, \(y = x^2\). First of all, what is the change in the area as \(x\) changes from 2 to 4? Draw a sketch of this situation, labeling \(dy\) and \(dx\). Do the same thing as \(x\) changes from 2 to 3, 2 to 2.5, 2 to 2.1, and 2 to 2.01. What do you observe? Try the same thing, but now start with \(x = 4\). The point is that the rate of change in the area, itself changes depending on the length. The function of \(x\) that expresses this relationship between the change in \(y\) and \(x\) is called the derivative, \(dy/dx\). Based on your experiments, what is the derivative of \(y = x^2\)?

10. The rate of damage reduction from armor is \(Reduction = Armor/(Armor + X)\) where \(X = 467 \times EnemyLevel - 22167\). What is the relative growing of damage reduction with Armor? With EnemyLevel?
Chapter 4

The Simplest Things

If you have studied Chapter 4 of CME, you will have learned the power rule for differentiating simple polynomial functions, known as terms. A term has the form:

\[ ax^n \]

with \( a \) being known as the coefficient of the term and \( n \) being known as the exponent of the term.

Here is the power rule for terms:

\[
\frac{d(ax^n)}{dx} = anx^{n-1}
\]

As example, consider the polynomial:

\[ y = 2x^3 \]

If the power rule is correct, then the derivative should be:

\[
\frac{dy}{dx} = 6x^2
\]

Let’s test. We start out by defining a function to represent the polynomial:

```sway
function y(x)
    { 2 * (x ^ 3); }
```

As before, the dependent variable becomes the name of the function and the independent variable becomes the formal parameter.

To learn more about storing code in files, see The Sway Reference Manual, Chapter 8.  

The derivative, or \( \frac{dy}{dx} \) can be written as this function:

1Typing large amounts of code into the interpreter can be rather painful. Thus for the rest of this book, even though interactions with the interpreter are shown, you should be storing Sway programs in files and running Sway programs from files.
function dy/dx(x)
{
  6 * (x ^ 2);
}

Is there any way to test whether or not the power rule is correct? Recall our revised ratio function from the previous chapter:

function ratio(y,x)
{
  var dx = 0.0001;
  var dy = y(x + dx) - y(x);
  dy / dx;
}

As you should remember, the ratio it computes is an approximation for the derivative at a given x. Note that we've changed the names of the formal parameters, w to x, dw to dx, and h to y, to reflect the fact that we are working with polynomials now.\(^2\)

We believe that ratio (for small dx) produces a number that should be close to what the function dy/dx produces.

Let's define a testing function:

function test(x)
{
  println("for x = ",x);
  inspect(ratio(y,x));
  inspect(dy/dx(x));
  println();
}

Given a value for x, the test function will print out the two results. Time for some nomenclature. When we use our ratio function, we are finding the derivative \textit{numerically}. That is to say, we use a small number for dx, find the resulting dy, and then compute the ratio \( \frac{dy}{dx} \). On the other hand, using the power rule (as embodied in the function \( dy/dx \)), we find the derivative \textit{symbolically} since we never actually compute an actual ratio.

\(^2\)In general, it does not matter what names are used for the formal parameters, so we could have left the ratio function untouched and it would still work exactly the same.
for $x = 30$
$ratio(y,x)$ is 5400.0001837
$dy/dx(x)$ is 5400

For this test at least, we see that there is good agreement between our numeric and symbolic solutions.

Clearly, though, the symbolic solution is preferred because:

- it is exact
- it is easier to compute

You may be asking yourself, can we write a program to find a derivative symbolically?

### 4.1 A Shortcoming to Our Approach

It certainly would be nice if we could use the symbolic approach to finding derivatives instead of our numeric approach using ratios. The problem is that our function representing a term:

```s
function y(x)
{
  2 * (x ** 3);
}
```

has the coefficient 2 and the exponent 3 hard-wired. If we are given such a function but aren’t privy to the internal details (such as the value of the hard-wired exponent), we cannot create the derivative function since we need to know the value of the exponent to do so and have no way to get at it. We need to contrive a way to retrieve the coefficient and the exponent from a function so we can use it to construct the derivative function.

The next section is to give you an exposure to one approach for extracting the components of a function. The approach uses *objects*.

To learn more about objects, see *The Sway Reference Manual*, Chapter 15.

OK, now that you are familiar with objects, we can begin to use objects to write a program that finds a derivative symbolically.

### 4.2 Polynomial Objects and the Power Rule

Instead of using a function to represent the polynomial $y$, we are going to use an object instead. From studying the reference manual, you know that an object is simply an environment and an environment is simply a table of variables and their values. You also know that you use a function to create an object and that a function that creates an object is conventionally known as a *constructor*. Finally, you know that to define a constructor function, you have that function return the predefined variable *this*.

Here’s a constructor for a simple polynomial object. We name the constructor *term*

```s
function term(a,n)
{
  function value(x)
```
Note that the constructor includes an internal function for computing the value of \( y \) given an \( x \), just as before. We call that function by using the \textit{dot} operator.

\begin{verbatim}
sway> var y = term(2,3);
OBJECT: <OBJECT 1671>
sway> y . value(4);
INTEGER: 128
sway> y . value(5);
INTEGER: 250
\end{verbatim}

Indeed, \( 2 \times 4^3 \) is truly 128 and \( 2 \times 5^3 \) is truly 250.

Now comes the good part! Storing a polynomial term as an object rather than a function allows us to extract the coefficient \( a \) and the exponent \( n \):

\begin{verbatim}
sway> y . a;
INTEGER: 2;
sway> y . n;
INTEGER: 3;
\end{verbatim}

We can even \textit{pretty print} the object \( y \) to see all its fields:

\begin{verbatim}
sway> pp(y);
OBJECT 2561:
  context: <OBJECT 749>
dynamicContext: <OBJECT 749>
callDepth: 1
constructor: <function term(a,n)>
this: <OBJECT 2561>
value: <function value(x)>
a: 2
n: 3
OBJECT: <OBJECT 2561>
\end{verbatim}

We see, among other fields, the correct values for \( a \) and \( n \). Later on, you will understand the meaning and purpose of the predefined fields: \textit{context}, \textit{dynamicContext}, and \textit{constructor}. The \textit{this} field you already know.

We want access to \( a \) and \( n \) because the power rule needs them to compute the derivative polynomial. Now we can write an expression that allows us to compute the derivative:

\begin{verbatim}
sway> var coeff = y . a;
sway> var exp = y . n;

sway> var z = term(coeff * exp, exp - 1);
\end{verbatim}
4.2. POLYNOMIAL OBJECTS AND THE POWER RULE

Of course, we can write a function to do this task for us:

```javascript
function powerRule(t)
{
    var coeff = t . a;
    var exp = t . n;
    term(coeff * exp, exp - 1);
}
```

Note that the `powerRule` function takes a `term` object as an argument and returns a new `term` object that represents the derivative. This is because the new term has a coefficient of \(a \times n\) and an exponent of \(n - 1\), just like the power rule dictates. Let’s check and make sure our power rule works as intended:

```sway
sway> var y = term(2,3);
sway> var z = powerRule(y);

sway> y . value(4);
INTEGER: 128
sway> z . value(4);
INTEGER: 96
```

Indeed it does!

Moreover, pretty printing `z` shows us the correct values for `a` and `n`:

```sway
sway> pp(dy/dx);
<OBJECT 2922>:
    context: <OBJECT 749>
    dynamicContext: <OBJECT 2808>
    callDepth: 2
    constructor: <function term(a,n)>
    this: <OBJECT 2922>
    value: <function value(x)>
        a: 6
        n: 2
    OBJECT: <OBJECT 2922>
```

You may not have noticed it, but we have a little lack of symmetry. While the term object has an internal function for computing its value, we use an external function, `powerRule` to compute its derivative. When we program using objects, we use internal functions when possible. So let’s rewrite the `term` function so that it can compute its own derivative:
function term(a,n)
 {
   function value(x)
   {
      a * (x ^ n);
   }
   function diff()
   {
      term(a * n,n - 1);
   }
   this;
 }

We name this new internal function, *diff*, to indicate taking the differential of our object. Note how we have dispensed with the variables *coeff* and *exp* inside the *diff* function and just used *a* and *n* directly.

Finally, we add a third internal function to *term*. This one is used to visualize our object in a more concise way than using the *pp* function. By convention, we will call this function *toString*:³

```
function term(a,n)
 {
   function value(x)
   {
      a * (x ^ n);
   }
   function diff()
   {
      term(a * n,n - 1);
   }
   function toString()
   {
      string(a) + "x^" + string(n);
   }
   this;
 }
```

The *toString* function converts both the coefficient and the exponent to strings and then concatenates those strings together to form a string representation of the term.

To learn more about strings, see *The Sway Reference Manual*, Chapter 18.

Here is *toString* in action:

```
sway> var y = term(2,3);
sway> var z = y . diff();

sway> y . toString()
STRING: "2x^3"

sway> z . toString()
STRING: "6x^2"
```

³The use of the name *toString* comes from the Java Programming Language, which uses a *toString* method for this very purpose.
Notice how much easier it is to pick out the coefficient and the exponent of a term by using the `toString` function. Another way to formulate the `toString` function takes advantage of the fact that if you add a string and a number (with the string on the left hand side of the plus sign, the number is automatically converted to a string. That, with the fact that Sway combines mathematical operators from left-to-right, lets us remove the string conversion of `n`:

```sway
def function toString()
    { string(a) + "^" + n; }
```

If we use an empty string to start the expression, we can remove the first string conversion as well:

```sway
def function toString()
    { "" + a + "^" + n; }
```

4.3 A Whole System of Objects

From here on out, we are going to develop a system for finding the derivative (and eventually, the integral) of many kinds of mathematical objects (limited only by our imagination and attention span). Each object in our system will do (at least) three things. The first is to compute its value at a given spot. The second is to compute its derivative (and eventually, its integral). The third is to compute its visualization. So all our mathematical objects will have `value`, `diff`, and `toString` internal functions or methods. 4

Even the simplest real-world mathematical objects such as numbers and variables will need to be re-created as Sway objects. That way, we can always take the derivative of an object without having to ask first if taking the derivative makes sense.

4.4 Questions

All formulas are written using Grammar School Precedence.

1. Why does the Sway implementation of $2x^3$ have parentheses:

   $$2 \ast (x \wedge 3)$$

2. Using the `powerRule` function, can you take the derivative of a derivative? Explain.

3. Write a function that uses the power rule function to answer the following questions on p. 58 of CME: 1, 2, 4, 6, 7. Name your function `p58`. Your function should calculate, then print the answers to the questions.

4. Using pencil and paper, work exercises 3, 8, 9, and 10 on p. 58 of CME.

---

4In the world of object-oriented programming (which is another way of saying programming with objects), these internal functions are called methods. We will use the term `method` from now one. Just remember, a method is simply an internal function.
Chapter 5

Constant Worries

So far, our polynomials contain only a single term, such as:

\[ y = 2x^3 \]

This is rather restrictive. For example, the equation of a line:

\[ y = mx + b \]

can be considered a polynomial whose term has power of one plus an added constant, \( b \). In other words:

\[ y = mx^1 + b \]

How do we deal with the constant? We could modify our \texttt{term} constructor to take a third argument, a constant. In this case, we would model the line:

\[ y = 3x - 5 \]

as:

\texttt{term(3,1,-5)}

While this would work, the approach is going to make things complicated when we start to model polynomials that are composed of multiple terms. Consider:

\[ y = 4x^4 - 2x^3 + 5x^2 + x - 3 \]

This polynomial could be represented by the following terms:

\texttt{term(4,4,0)}
\texttt{term(-2,3,0)}
\texttt{term(5,2,0)}
\texttt{term(1,1,-3)}
with the first three terms having zero as a constant. Alternatively, we could represent the polynomial this way instead:

\[
\begin{align*}
\text{term}(4,4,1) \\
\text{term}(-2,3,2) \\
\text{term}(5,2,3) \\
\text{term}(1,1,-9)
\end{align*}
\]

Now every term has a constant (note all the constants add up to -3 as they should). You should see two problems now with adding a third argument to \text{term}. The first is most terms don’t have (or shouldn’t have) a constant. The second is it is unclear which term, in a collection of terms, should hold the constant.

Before I give you a better way, I’m going to tell you a story first. A friend of mine had a brother (or cousin or whatever, I don’t remember which) who, lacking a crowbar, used a Craftsman screwdriver to pry some rather heavy and rather stuck piece of machinery out of its housing. The piece refused to budge so the brother (or cousin or whatever) put his full weight on the screwdriver, which promptly bent. Now you may not know this, but a screwdriver that is bent is nearly useless for doing things a screwdriver normally does (like screwing or unscrewing screws). As it happens, Craftsman hand tools had a lifetime warranty, so the brother (or cousin or whatever) took the bent screwdriver back to Sears and asked for a replacement. The clerk happily obtained a new screwdriver and remarked, as she exchanged screwdrivers, “You know, Sears sells Craftsman crowbars as well”.

What is the moral of the story? It is, of course, \textit{Use the right tool for the right job}. \footnote{Sometimes, Computer Scientists ignore this rule, preferring an alternate the moral of the story: \textit{Next time, use a bigger screwdriver}. This is because Computer Scientists (again with the laziness) don’t want to write something from scratch (a crowbar), but would rather use something else that has already been written and can be adapted for the job at hand (the screwdriver). Sometimes, this works nicely, but sometimes it leads to added complications down the road. In this particular case, a constant \( c \) could be represented as a term with coefficient \( c \) and exponent zero. Instead, we will design our second kind of object in our calculus system.} We need a new tool to represent constants.

### 5.1 Constant Objects

You are probably asking, what’s the big deal with constants anyway? Why can’t we just use numbers? We could, but consider this exchange:

\[
\text{sway}> \text{var } s = t . \text{diff}();
\]

\text{OBJECT: <OBJECT 4369>}

Suppose \( t \) is bound to a term object. Then all is well and good; \( s \) is bound to a term object as well. But suppose \( t \) is bound instead to the number eight. We get a much nastier response from the terminal:

\[
\text{sway}> \text{var } t = 8
\]

\[
\text{sway}> \text{var } s = t . \text{diff}();
\]

\text{EVALUATION ERROR: :accessError}

\text{dot operator used on type INTEGER}

We tried to treat the number eight as an object, which it is not. Since numbers are valid mathematical entities and since it is sensible to take the differential of a number, we are left with two choices:

- Everytime we wish to take the differential or find a value or visualize, we need to test to see if the entity we are dealing with is an object or not.
• Make sure numbers are objects with `diff`, `value`, and `toString` methods.

The first choice means every bit of code we write in the future has to take into account that we have two kinds of things to deal with and we best not forget it.

The second choice means spending a little time up front, but never worrying differing kinds of entities again.

Clearly the second choice is the way to go. Before we write our constructor for representing constants, we need to ask ourselves three things:

• What is the value of a constant at a particular point?
• What is the derivative of a constant?
• What is a reasonable string representation of a constant?

These three things correspond to the three methods all of our constructors must have.

The first of these questions seems to be a bit strange. The value function for a `term` computed a value of the term given a value of \( x \). As \( x \) varies, the value of the term varies as well. But for a constant, there is does not seem to be an \( x \). This is not exactly true, since there is an equation where the value of \( y \) remains constant regardless of the value of \( x \). An example is the equation \( y = 5 \). We can plot that equation:

![Graph of y = 5](image)

and we see that, regardless of the value of \( x \), the value of \( y \) remains constant. Thus, the `value` method of a constant object should ignore the given value of \( x \) and return its constant value:

```javascript
function constant(n)
{
    function value(x)
    {
        n;
    }
}
```
Note how the value method always returns $n$:

```python
sway> var c = constant(5);
sway> c . value(0);
INTEGER: 5
sway> c . value(3);
INTEGER: 5
```

Moving on, what is the differential of a constant? In other words, how does the value of our object change as the value of $x$ is changed. We see from the interaction above that the value doesn’t change at all. In other words the change is zero. You might, therefore be tempted to write the `diff` method for a constant object as:

```javascript
function diff()
{
    0;  //incorrect
}
```

This would be incorrect, however, since all `diff` methods must return objects that have `diff` methods themselves. We saw that with terms: the `diff` method of a term object returns a term object, which has a `diff` method as desired. For the task at hand, what object should be used to represent the number zero? A constant object!

```javascript
function constant(n)
{
    function value(x) { n; }
    function diff()
    {
        constant(0);
    }
    this;
}
```

Our final question to answer is how do we visualize a constant? That is easy to answer: we simply convert the number into a string. We are left with our complete constant constructor:

```javascript
function constant(n)
{
    function value(x) { n; }
    function diff() { constant(0); }
    function toString()
    {
        "" + n;
    }
    this;
}
```
5.2. USING CONSTANT OBJECTS

Let’s create a constant with value 7 and see if indeed the resulting value of the object is constant regardless of the value of \( x \).

```sway
var c = constant(7);

sway> c . value(10);
INTEGER: 7
sway> c . value(1000);
INTEGER: 7
```

Confirmed.

5.2 Using Constant Objects

That was quite a detour! Remember, we are trying to model a line using \( \text{term} \) and \( \text{constant} \). The line:

\[
y = mx + b
\]

can be represented by two objects:

```sway
\text{term}(m,1)
\text{constant}(b)
```

What happens when we add them together?

```sway
sway> var m = 3;
sway> var b = -5;

sway> \text{term}(m,1) + \text{constant}(b);
EVALUATION ERROR: :argumentTypeError
addition: cannot add type OBJECT to type OBJECT
```

What we need now is a way to glue these two objects together. The next chapter will show us how.

5.3 Questions

All formulas are written using Grammar School Precedence.

1. Use Sway to represent the individual terms of the following polynomial \( y = 6 + 3x - 5x^2 + x^3 \). What is the value of this polynomial for \( x = 2 \).

2. Use Sway to represent the individual terms of the following polynomial \( y = 6 + 3/x - 5/x^2 + 1/x^3 \). What is the value of this polynomial for \( x = 2 \).

3. Infant mortality is inversely proportional to income level. What is the relative growing of infant mortality with income level?
Chapter 6

Guzzintas and other Cipherin’

Although we now have a nifty way to represent constants, we are now left with a bigger problem: how to combine terms and constants. We can’t use Sway’s + operator, since it only works for numbers and strings.\(^1\) Instead, we will create an object to hold the two items to be added.\(^2\) Let’s create a constructor named plus to remind us that it generates an object that holds the (potential) sum of two objects:

```sway
function plus(p,q) //p and q are terms/constants
{
    this;
}
```

We can’t do much with plus objects, however, except extract the original items, bound to \(p\) and \(q\).

How should we enhance our plus constructor? Just like terms and constants, a plus object should be able to:

- compute its value
- compute its derivative
- visualize itself

Of course, the plus object doesn’t know how to do any of these things, since it’s just a container for holding two items. What it can do, however, is ask those items to perform computations and then combine the results in a meaningful way.\(^3\)

Now our questions become:

- how should the values of the items be combined?

---

\(^1\) Later, we will learn how to override the + operator so that it can add terms and constants as well.

\(^2\) Why we are doing this is not obvious. As we proceed, we will see that the approach works, but it won’t give us much insight on how to come up with such solutions in the first place. We are essentially going to delay the addition of the items until we really need to add them together (e.g., when we are trying to compute a specific \(y\) value). The idea of delaying is a powerful computer science concept. Knowing when to delay, however, is more art than science.

\(^3\) There are two main kinds of relationships between objects. In this case, the relationship between the plus object and \(p\) (or \(q\)) is clientship. The object \(p\) (or \(q\)) is said to be a client of the plus object since \(p\) (or \(q\)) is a component. The other main relationship between objects in inheritance in which objects share traits. You can read more about in [[The Sway Reference Manual]]. Although explicit inheritance is not used in this case, one can think of terms and constants inheriting the idea of the three methods: value, diff, and toString.
• how should the differentials of the items be combined?

• how should the visualizations of the items be combined?

For the value method, how should the value of $p$ and the value of $q$ be combined? Since the value of $p$ is a number and the value of $q$ is a number and since the plus object represents the additions of its items, we can simply add the item values together:

```plaintext
function plus(p, q) // p and q are terms/variables,
{ // constants
    function value(x)
    {
        p . value(x) + q . value(x);
    }
    this;
}
```

Thus, to compute the value of a two terms added together, we simply find the values of the terms separately, then add the results together. Mathematically,

$$v(p + q) = v(p) + v(q)$$

where $v$ represents a function for ‘finding the value of’. Let’s test:

```sway
sway> var a = term(-5,2);
sway> var b = term(7,0);
sway> a . value(3) + b . value(3);
INTEGER: -38
sway> var z = plus(a, b);
sway> z . value(3);
INTEGER: -38
```

Note that our plus constructor doesn’t actually specify what kinds of objects the formal parameters $p$ and $q$ must be bound to; the only requirement is that those objects have a value method that takes a single argument. We will take advantage of this fact later and use plus objects to glue together terms, constants, and other plus objects, nilly willy.

### 6.1 Differentiating Sums

We now turn our attention to the second method: plus objects need to implement the diff method. Of course, the power rule won’t work for sums, because the rule only works only for computing the derivative of a single term.

It turns out the derivative of two things added together is the sum of the derivatives of each thing alone. Mathematically,

$$\frac{d(a + b)}{dx} = \frac{d(a)}{dx} + \frac{d(b)}{dx}$$
Now here’s the tricky part (well, it may seem tricky now, but in a short time, you will think this is as easy as pie). We use a plus object to represent the addition of two terms, correct? The right-hand-side of the equation above involves an addition. What are being added? Two derivatives. If \( a \) and \( b \) are term objects, then what kinds of things are the derivatives of \( a \) and \( b \). Both are terms! Thus the right-hand-side in this case is simply the sum of two terms. But what do we use to represent the sum of two terms......wait for it......a plus object.

Wow! Then the derivative of a plus object, consisting of two terms, is another plus object containing the derivatives of those two terms. This is both amazingly powerful and amazingly simple at the same time.

If the above explanation has confused you, perhaps looking at the code will help. Here is the revised plus:

```javascript
function plus(p,q) //p and q are calculus objects
{
    function value(x) { p . value(x) + q . value(x); }
    function diff()
    {
        p . diff() plus q . diff();
    }
    this;
}
```

To differentiate plus objects, we use the two formal parameters, \( p \) and \( q \), take their derivatives, and then combine the results into a plus object.

### 6.2 More than two terms

Our plus constructor is fine for representing lines as a line is composed of two terms, one of them being a constant. What do we do for a sum of three or more terms?

How about doing...nothing.

Yes, nothing. It turns out that our design of plus is so fine it not only handles the sum of two terms but the sum of a plus object and a term.\(^4\) Recall that the only requirement for the parameters \( p \) and \( q \) in a plus object are that they both be bound to objects that have `value` and `diff` methods. Does a plus object have these methods? Yes, indeed! So that means \( p \) or \( q \) or both can be bound to plus objects. Let’s see:

```javascript
var a = constant(-5);
var b = plus(term(3,1),a);
var y = plus(term(4,2),b);
```

The variable \( y \) now refers to the polynomial:

\[
y = 4x^2 + 3x^1 + -5
\]

Now, just because we should (and can) be able to combine sums and terms nilly-willy with our plus constructor doesn’t mean we can assume plus is completely correct as written. We need to test \(^5\) in order to convince ourselves thoroughly that the code works. Let’s test \( a \), \( b \), and \( c \) at some value of \( x \) that is easy to verify, say \( x = 2 \):

---

\(^4\)You will find these happy occurrences often if you write code that is simple and elegant.

\(^5\)When you are a senior, rather than test some code to see if it appears correct, you might prove it correct. The advantage of proving code correct is that performing all possible tests is sometimes difficult or impossible.
sway> a . value(2);
INTEGER: -5

sway> b . value(2);
INTEGER: 1

sway> y . value(2);
INTEGER: 17

In this interaction, we see that $a$, which is a calculus object representing the constant $-5$, has a value of -5 when $x$ has a value of 2. The object $b$, which is a plus object representing $3x^1 + -5$, has a value of 6 - 5 or 1. The object $y$, which is also a plus object and represents $4x^2 + 3x^1 + -5$, has a value of 16 + 6 - 5 or 17.

It is not surprising that the object $a$, which is just a constant, and the object $b$, which is composed of two terms, both give the correct answer; we are using their constructors exactly in the manner they were intended. Less obvious is why $y$ works, being composed of a term and a plus object. The next section will demonstrate visualization to help you understand ‘why’.

### 6.3 Visualizing sums

A very good way to see why $y$ works is to visualize the $y$ object. Both terms and objects have toString methods for visualizing themselves. We need to add such a method to plus. Earlier, we discovered that the value of a sum was the sum of the values. We also discovered the derivative of a sum is the sum of the derivatives. Now we need to visualize a sum. How do we do that? If you are thinking, to visualize a sum, maybe we need to sum the visualizations, you can move to the head of the class! We begin by modifying the plus constructor to add a toString method.

```javascript
function plus(p,q) //p and q are calculus objects
{
    function value(x) { p . value(x) + q . value(x); }
    function diff() { p . diff() plus q . diff(); }
    function toString()
    {
        p . toString() + " + " + q . toString();
    }
    this;
}
```

Note that plus visualizes itself by using the visualizations of its component objects and adds a ‘+’ sign to indicate addition.

Now we remake $a$, $b$, and $y$ with these new definitions of term and plus in force:

```javascript
var a = constant(-5);
var b = plus(term(3,1),a);
var y = plus(term(4,2),b);
```

Let’s see what $y$ looks like:

```
sway> y . toString();
STRING: 4x^2 + 3x^1 + -5
```
6.4 Testing differentiation

Looks exactly as intended.

To see further why the visualization works, it is sometimes useful to look at the sequence of calls generating the result. These calls are organized into what is known as a ‘call tree’. Actions within a call are indented one-level from the originating call. Leftward pointing arrows represent return values.

Here is to call tree for y’s toString method call:

```
call y . toString()       //object y is plus(term(4,2),b)
call p . toString()       //object p is term(4,2)
  "4x^2"                //return value
  " + 

call q . toString()       //object q is b, plus(term(3,1),a)
call p . toString()       //object p is term(3,1)
  "3x^1"                //return value
  " + 

call q . toString()       //object q is a, constant(-5)
  "-5"                //return value
  "4x^2 + 3x^1 + -5"    //return value
```

Combining the strings from top to bottom at the first indentation level gives us the overall return value:

```
4x^2 + 3x^1 + -5
```

We can construct a call tree determining the value of y when x = 2, as well:

```
call y . value(2)
call p . value(2)   // p is 4x^2
  16
  +
call q . value(2)   // q is 3x^1 + -5x^0
  6
  +
call q . value(2)   // q is -5
  -5
  +
  1
  17
```

Summing the return values at the first indentation level yields 17, the overall return value.

Visualization is an important technique. You should always include a visualization method for all your objects so that you can easily debug your program when things go awry. In such cases, your visualization will likely point out that an object does not appear as you expect it to, an important first step in solving your problem.

6.4 Testing differentiation

We have tested our plus constructor by computing its value and visualization, but is its derivative function correct? Let’s find the derivative of the polynomial bound to the variable y previously. Recall y visualized as:
We would expect the \textit{diff} function to produce a polynomial that visualizes to something like:

$$8x + 3$$

Let’s see...

```javascript
var dy/dx = y . diff();

sway> dy/dx . toString();
STRING: 8x^1 + 3x^0 + 0
```

Looking at the last portion of the visualization, we note that zero plus anything is the anything, so the result is equivalent to:

$$8x^1 + 3x^0$$

Noting, as before, that \(x^0\) is equal to 1, the result becomes:

$$8x^1 + 3*1$$

or

$$8x^1 + 3$$

Finally, realizing that \(x^1\) is simply \(x\), the result becomes:

$$8x + 3$$

as desired. Some of the questions at the end of this chapter explore how to have the \textit{plus} and \textit{term} perform some of these simplifications automatically.

### 6.5 Other mathematical combinations of terms

What if we wish to subtract two terms, as in:

$$y = 3x^2 - 4x$$

While figuring out the \textit{value} and \textit{toString} methods of a minus constructor takes little effort, what is the rule for finding the derivative of a subtraction of items? It is similar, but not quite the same as the rule for the addition of items:

$$\frac{d(a - b)}{dx} = \frac{d(a)}{dx} - \frac{d(b)}{dx}$$
In English, the differential of a subtraction is the subtraction of the differentials. Namely, we subtract the differentiation of the subtrahend (the right operand) from the differentiation of the minuend (the left operand). Now we have all the information we need to define \textit{minus}:

\begin{verbatim}
function minus(p,q)
{
    function value(x) { p . value(x) - q . value(x); }
    function toString() { p . toString() + " - " + q . toString(); }
    function diff() { p . diff() minus q . diff(); }
    this;
}
\end{verbatim}

This is just like \textit{plus}, except that minus signs are used instead of plus signs in methods \textit{value} and \textit{toString} and \textit{minus} is used instead of \textit{plus} to join the two derivatives.\footnote{A fine example of \textit{programming by analogy}. Be aware that making a new function by copying an existing function and making minor changes is usually a ‘bad idea’. If you find yourself doing this, ask yourself how can the the two functions be combined into a single function? There is a nice way to do this for \textit{plus} and \textit{minus}, but we’ll skip that as to not impede the narrative flow.}

What about multiplying and dividing terms? According to CME, the mathematical rule for differentiating a product is:

\[
\frac{d(a \cdot b)}{dx} = (a \cdot \frac{db}{dx}) + (b \cdot \frac{da}{dx})
\]

The rule for division is more complicated still:

\[
\frac{d(a/b)}{dx} = \frac{(b \cdot \frac{da}{dx}) - (a \cdot \frac{db}{dx})}{b^2}
\]

The implementations of the \textit{diff} methods for \textit{times} and \textit{div} constructors are left as an exercise.

\section*{6.6 Questions}

All formulas are written using Grammar School Precedence.

1. What happens if you pass a \textit{it plus object to the original \textit{powerRule} function? Explain what happens.
2. Define and test a \textit{times constructor}. Be sure to add \textit{value}, \textit{toString} and \textit{diff} methods.
3. Define and test a \textit{div constructor}. Be sure to add \textit{value}, \textit{toString} and \textit{diff} methods.
4. Modify the \textit{term} return a constant object representing zero instead of \textit{this} if a zero is passed in as the coefficient.
5. Modify the \textit{term} return a constant object representing the coefficient instead of \textit{this} if a zero is passed in as the exponent.
6. Modify the \textit{toString} method of \textit{term} to ignore the coefficient and/or the exponent if it is 1. That is, \textit{term(3,1)} should display as 3x, not 3\textit{x}^1, and \textit{term(1,4)} should display as \textit{x}^4, not 1\textit{x}^4.
7. Modify the \textit{plus} constructor to throw an argument if it is a constant zero. How do you ‘throw away’ an argument?
8. Modify the `plus` constructor so that if the second argument is a term with a negative coefficient or a negative constant, it returns an appropriate `minus` object instead of `this`.

9. Use Sway for Problem 6 on page 64 of CME.

10. Use Sway to differentiate \( y = (x - \frac{3}{2}) + (x^2 - 5x) + (3x^3 + 7x^2 + 3x + 5) \). Define function \( y \) and work exercises 1–5 on p. 64 of CME using pencil and paper.

11. Use pencil and paper to solve exercise 11 on p. 77.
Chapter 7

Lather, Rinse, Repeat

As CME, Chapter 7 notes, suppose:

\[ y = f(x) \]

That is to say, \( y \) is some function of \( x \). If that is so, then:

\[ \frac{dy}{dx} \]

is sometimes denoted:

\[ f'(x) \]

with the apostrophe indicating differentiation.

Because of Sway’s liberal views on naming variables, we can use the same notation. Consider defining a line:

\[ \text{var } f = \text{line}(3,-5); \quad \text{//equivalent to } y = 3x - 5 \]

where the \text{line} constructor is just a wrapper for a call to the \text{plus} constructor:

\[
\text{function line}(m,b) \\
\{ \\
\quad \text{term}(m,1) \text{ plus constant}(b); \\
\}
\]

Then, we can differentiate our line thusly:

\[ \text{var } f' = f . \text{diff}(); \]

Since \( f \) represents a line and the differentiation of a line yields the slope of a line and the slope of the line is always a constant \( m \), we would expect that evaluating \( f' \) at different points would always yield the slope:
If we repeat the differentiation process again, we find out how fast the slope is changing at any given point. For a line, the slope never changes so we would expect the derivative of the derivative (also known as the second derivative) always to yield zero.

```javascript
var f'' = f'. diff();
```

```javascript
sway> f''. toString();
STRING: 0x^-1 + 0
sway> f''. value(0);
INTEGER: 0
sway> f''. value(5);
INTEGER: 0
```

Confirmed. There is a problem lurking, however. The visualization of $f''$ yielded:

$$0x^{-1} + 0$$

If we had tried to find the value of $f''$ at zero, we would have gotten a divide by zero error. We will fix this later.

### 7.1 More on visualization

We really need to do something about our visualization. It’s printing out terms that we really don’t need to see. Let’s simplify the output by making term’s `toString` method more complicated. Please implement the following rules for term’s `toString` method:

1. if both the coefficient and the exponent are 1, it `toString` should return "x"
2. if the exponent is one, it `toString` should return " + a + "x"
3. if the coefficient is one, it `toString` should return "x" + n
4. otherwise, it `toString` should return " + a + "x" + n

Earlier rules take precedence over later rules. After implementing these rules and remaking $f$, $f'$, and $f''$, we get the following visualizations:

```javascript
sway> f . toString();
STRING: 3x + -5
sway> f' . toString();
STRING: 3 + 0
sway> f'' . toString();
STRING: 0 + 0
```

Somewhat better. Now we need to improve `plus` to throw away constant terms with a value of zero. We can do this in the main body of the `plus` constructor:
1. if the first argument is equivalent to zero, have it plus return the second argument
2. if the second argument is equivalent to zero, have it plus return the first argument
3. otherwise, have it plus return it this

Your logic should look like this:

```javascript
function plus(p, q)
{
    function value(x) { ... }
    function toString() { ... }
    function diff() { ... }
    if (p is equivalent to zero) //pseudocode
    {
        q;
    }
    else if (q is equivalent to zero) //pseudocode
    {
        p;
    }
    else
    {
        this;
    }
}
```

How do we determine if an argument to `plus` is equivalent to zero? Since we are representing zero as a constant with a zero value, we could use logic like this:

```javascript
if (p is :constant && p . value(0) == 0)
```

Note that we could pass any number to the value method since a constant will return its true value regardless. Our plus constructor becomes:

```javascript
function plus(p, q)
{
    function value(x) { ... }
    function toString() { ... }
    function diff() { ... }
    function isZero(x) { p is :constant && p .value(0) == 0; }
    if (isZero(p)) { q; }
    else if (isZero(q)) { p; }
    else { this; }
}
```

Note the addition of the `isZero` function to make the code more readable. With these changes and after remaking $f$, $f'$, and $f''$ with the new versions of `plus`, `term` and `constant`, we get the following visualizations:
7.2 Successive Differentiations of Complex Polynomials

It’s rather uneventful to repeatedly differentiate a line. Higher-order polynomials\(^1\) are a little more interesting. Rather than build up a high order polynomial of many terms piece by piece, as in:

```javascript
var a = term(1,0);
var b = plus(term(2,1),a);
var c = plus(term(3,2),b);
var y = plus(term(4,3),c);
```

We can build it up *en masse*:

```javascript
var y = plus(term(4,3),plus(term(3,2),plus(term(2,1),term(1,0))));
```

or since this whole scale building can be hard to read, we can build \(y\) using infix operator syntax:

```javascript
var y = term(4,3) plus term(3,2) plus term(2,1) plus term(1,0);
```

No matter how you make \(y\) (it’s all a matter of preference), we can differentiate it successively:

```javascript
var y' = y . diff();
var y'' = y' . diff();
var y''' = y'' . diff();
```

To check numerically, let’s evaluate \(y''\) at \(x = 1\) and \(2\).

```javascript
sway> y'' . value(1);
INTEGER: 30
sway> y'' . value(2);
INTEGER: 54
```

\(^1\)The higher the order of a polynomial, the larger the largest exponent in the terms making up the polynomial.
The object $y'''$ should represent the constant 24:

```sway
sway> y'''. value(1);
INTEGER: 24
sway> y'''. value(2);
INTEGER: 24
```

Everything looks good.

### 7.3 Questions

1. Justify mathematically each of the simplifying visualization rules for a term.

2. Add simplification to the `minus` constructor. The simplification is similar to `plus` but is a tiny bit trickier.

3. Add simplification to the `times` constructor. You should return constant zero if either argument is zero and if one of the arguments is constant one, return the other. Otherwise, return `this`.

4. Add simplification to the `div` constructor. You should return constant zero if the numerator (the first argument) is constant zero. You should return the numerator if the denominator (the second argument) is constant one. Otherwise, you should return `this`.

5. Represent and plot $y = 2x^2$ in sway using `gnuplot`. What is $dy/dx$? Plot it versus $x$. What does it represent? What is $d^2y/dx^2$? Plot it versus $x$. What does it represent? What is $d^3y/dx^3$? Plot it versus $x$.

6. Use both sway or pencil and paper to do 1-3 on p. 82.
Chapter 8

As Time Goes By

In CME, chapter 8, SPT discusses how Newton used a dot over a variable to indicate differentiation while Liebniz used the notation in this book: $\frac{dy}{dx}$. The advantage to Liebniz’s notation is that it explicitly states which independent variable is to be considered when differentiation occurs (in case there is more than one independent variable).

For example, if

$$y = 3x^2 - 5$$

then

$$\frac{dy}{dx} = 6x$$

On the other hand, differentiating with respect to $t$ instead of $x$ gives:

$$\frac{dy}{dt} = 0$$

since $3x^2 - 5$ is considered a constant with respect to the independent variable $t$. That is to say, no matter how much you change the value of $t$, the value of $3x^2 - 5$ doesn’t change (since you are not changing $x$, just $t$).

We have been assuming, up until now, that the independent variable and the variable with respect to which we take derivatives are one and the same.

Let’s investigate what our code might look like if we did not make that assumption.

8.1 An unassuming differentiation system

Based upon our visualization for the term constructor, we have been assuming the independent variable is $x$, because we end up with visualization like:

$$3x^2$$

1As SPT points out, the dot notation is used with the assumption that differentiation is performed with respect to time $t$. Hence, the name of this chapter.
If we don’t hard-wire the independent variable, we will need to pass it in. Here is a skeleton of a version of \textit{term} where the name of the independent variable is to be passed in:

\begin{verbatim}
function term(a,iv,n)
 {
     function value(x) { ... }
     function toString() { ... }
     function diff(wrtv) { ... }
     this;
 }
\end{verbatim}

The first major difference between the new version of \textit{term} and the old version is that the new \textit{term} has three formal parameters, instead of two. The second formal parameter, \textit{iv}, which stands for the \textbf{independent variable}, represents the independent variable. We presume it will be bound to symbols such as \textit{x}, \textit{t}, and the like. We see this in the basic \textit{toString} method (the one with no simplifications) which changes from:

\begin{verbatim}
function toString() { \\
    "" + a + "x" + n;
}\end{verbatim}

to:

\begin{verbatim}
function toString() { \\
    "" + a + iv + "" + n;
}\end{verbatim}

Note that \textit{x} in the original version was part of a string and thus fixed. In the second, \textit{iv} is a variable that is (or rather will be) bound to a symbol. Looking at a visualization will help us sort things out:

\begin{verbatim}
var t = term(5,:w,3);

sway> t . toString();
STRING: 5w^3
\end{verbatim}

The second major change is that the differentiation method, \textit{diff}, now takes an argument, \textit{wrtv}, which stands for the \textbf{with-respect-to variable}. This is the variable with respect to which differentiation should proceed. If the independent variable and the with-respect-to variable are the same, differentiation proceeds as before. If not, a constant zero is generated:

\begin{verbatim}
function diff(wrtv)
 {
     if (wrtv == iv)
     {
         term(a * n,iv,n - 1);
     }
     else
     {
         constant(0);
     }
 }
\end{verbatim}

Now let’s test:
8.2. WHAT ABOUT THE VALUE METHOD?

We do not need to change the value method for terms. This is because a value for the independent variable is used, rather than its name. In other words, the value method performs a numeric calculation, while the diff method performs a symbolic calculation. Thus, value needs a number and diff needs a name.

Since both toString and diff use the name of a term variable rather than a value, they both needed modification because names are not longer hard-wired.

8.3 Questions

1. What happens if we rename the formal parameter iv to be x and replace every occurrence of iv with x. Explain.
2. Redefine the simplifying term constructor so that it does not assume the independent variable to be x.
3. Redefine the simplifying sum constructor so that it does not assume the independent variable to be x.
4. Complete the unassuming differentiation system (minus, times, and div).
5. Using Sway, solve 2 on p. 92.
6. Using Sway, solve 4 and 5 on p. 92.
7. We are doing a game to throw a beanbag into a trash can and we want to model the physics correctly. The horizontal displacement is $x = x_0 + v_{x0} t$ and the vertical displacement is $z = z_0 + v_{z0} t - \frac{1}{2}gt^2$. What is change of both $x$ and $z$ with $t$? Let $x_0 = 0$, $v_{x0} = 3$, $z_0 = 5$, and $v_{z0} = 3$. Plot $x$ and $z$, and the velocity in $x$ and $z$ directions versus time from 0 to 10.
Chapter 9

Working on the Chain Gang

Sometimes, as SPT in CME, Chapter IX, points out, you find yourself puzzling over how to differentiate something complicated like:

\[ y = 3(x^2 + 17)^2 \]

The approach is to make the expression simpler by abstracting away the detail. Let \( a \) be the polynomial:

\[ a = x^2 + 17 \]

We can represent this using a sum of terms:

Now, \( y \) can be rewritten as:

\[ y = 3a^2 \]

To find the derivative of \( y \) with respect to \( x \), we use the chain rule:

\[ \frac{dy}{dx} = \frac{dy}{da} \times \frac{da}{dx} \]

In other words, the differential of \( y \) with respect to \( x \) is equal to the differential of (the rewritten) \( y \) with respect to (the new) \( a \) multiplied by the differential of (the new) \( a \) with respect to \( x \).

Programmatically, we have:

```swift
var a = term(1,:x,2) plus constant(17);
var y = term(3,:a,2);

var dy/dx = y . diff(:a) times a . diff(:x);
```

Looking at what \( dy/dx \) represents, we have:

```
sway> a . toString();
```

55
CHAPTER 9. WORKING ON THE CHAIN GANG

STRING: \( x^2 + 17 \)
sway> y . toString();
STRING: \( 3x^2 \)
sway> dy/dx . toString();
STRING: \( 6a * 2x \)

Doing the final substitution by hand \((a = x^2 + 17)\), we get the final answer for \(dx/dy\):

\[
dy/dx = 6 \cdot (x^2 + 17) \cdot 2x \\
= (6x^2 + 102) \cdot 2x \\
= 12x^3 + 204x
\]

9.1 Implementing the chain rule

It would be nice to have the chain rule substitution step automatically done for us, but doing so requires a bit of work, both conceptually and programmatically. We begin by extending the idea of abstracting the variable of a term. Recall that, at first, we hard-wired the term variable as \(x\). Next, we allowed the caller of the \texttt{term} constructor to pass in the independent variable as a Sway symbol. The next step in the abstraction is to allow the term variable to be a term (or sum of terms or whatever) itself. If we did so, then a term’s \texttt{diff} method would become the chain rule:

\[
\text{function diff(wrtv)} \\
\{ \\
\text{term(a \ast n,iv,n - 1) times iv . diff(wrtv);} \\
\}
\]

Obviously, the independent variable \(iv\) can no longer be a symbol, but must be instead an object with a \texttt{diff} method. So, to represent a term of the form:

\[ax^b\]

we would need to use an object to represent the variable \(x\). The constructor for such a variable object would look similar to \texttt{term} and \texttt{plus} constructors. That is, it must have \texttt{value}, \texttt{toString}, and \texttt{diff} methods.\(^1\)

\[
\text{function variable(name)} \\
\{ \\
\text{function value(x) \{ x; \}} \\
\text{function toString() \{ "" + name; \}} \\
\text{function diff(wrtv)} \\
\{ \\
\text{if (wrtv == name) \{ constant(1); \} else \{ constant(0); \}} \\
\}
\]

\(^1\)This is the heart of the object-oriented approach to programming: related objects have the same methods, but the methods are customized for the particular object.
9.1. IMPLEMENTING THE CHAIN RULE

The rule for finding the derivative of a simple variable is: if the with-respect-to variable matches, the result is one. If not, the result is zero. We will use constants to represent the numbers zero and one; in this way, every item in our system, including numbers, has `toString`, `value`, and `diff` methods.

To make our lives simpler, we can add the following logic to the body of the `term` constructor. If a symbol is passed in as the independent variable, we will convert it into a variable object. In this way, we can pass in a symbol as before. Here is a mock-up of the new term constructor:

```javascript
function term(a, iv, n)
{
    function value(x) { ... }
    function toString() { ... }
    function diff(wrtv)
    {
        term(a * n, iv, n - 1) times iv . diff(wrtv);
    }

    iv = toCalculus(iv);
    if (a == 0) { return constant(0); }
    if (e == 0) { return constant(a); }
    this;
}

function toCalculus(x)
{
    if (x is :SYMBOL, variable(x), x);
}
```

We define a new function, `toCalculus`, to handle the conversion from symbol to calculus variable in anticipation of converting other non-calculus objects to calculus ones. See the exercises at the end of the chapter for more on this. Note that in addition to converting an independent variable given as a symbol via `toCalculus`, this version of term checks for zero coefficients and zero exponents, returning an appropriate constant in those cases.

We will also need to modify term’s `toString` method to call `iv`’s visualization. Here is the new non-simplifying version:

```javascript
function toString()
{
    "" + a + iv . toString() + "^" + n;
}
```

Let’s test our modified system:

```javascript
var t = term(4,:x,3);
var t' = t . diff(:x);
```

This little trick illustrates an important principle in the design of the computer programs: do as much for the user of your code as possible. We could force the user to pass in a variable object or we could allow the user to pass in a symbol, as before, and do the work ourselves.
It seems to be working so far for simple variables. Now let’s try our original problem:

\[ y = 3(x^2 + 17)^2 \]

First, we make our polynomial:

```javascript
var a = term(1,:x,2) plus constant(17);
var y = term(3,a,2); // not :a
```

Now, we visualize it:

```javascript
sway> y . toString();
STRING: 31x^2 + 17^2
```

Close, but not quite correct. What did we do wrong? We need to parenthesize the visualization of \( iv \):

```javascript
function toString()
{
  "" + a + "(" + iv . toString() + ")" + "" + n;
}
```

Remaking \( a \) and \( y \) with term’s new visualization yields:

```javascript
var a = term(1,:x,2) plus term(17,:x,0);
var y = term(3,a,2);
```

```javascript
sway> y . toString();
STRING: 3(1x^2 + 17)^2
```

If you use a simplifying `toString` method for terms, you should get:

\[ 3(x^2 + 17)^2 \]

exactly as desired! Now let’s differentiate \( y \) (you will need your `times` constructor up and running):

```javascript
var y' = y . diff();
```

```javascript
sway> y' . toString();
STRING: 6(x^2 + 17) * 2x
```
9.2 Questions

1. Explain why the it diff method for terms no longer needs to test whether or not the with-respect-to variable matches the independent variable.

2. Modify the toCalculus function so that it also converts integers and reals into appropriate calculus objects.

3. Implement the one function.

4. Modify the simplifying toString method for terms to print out parentheses only when the independent variable is complex and either the coefficient or the exponent is not equal to one. *Hint:* Create a term method that adds parentheses around iv's visualization if it is complex but simply returns iv's visualization if it is not. Call this method from toString where appropriate.

5. CME p. 100, 1, 8 using sway

6. CME p. 100, 2, 3, 5, 8 using pencil and paper
Chapter 10

Slippery Slopes

In Chapter X of CME, SPT states that the derivative (at a given point) can be viewed as the slope of the tangent to a curve (at that point). Let’s use our software to examine slopes.

Consider parabolas of the form:

\[ y = (x - h)^2 + v \]

The \( h \) variable induces a rightward shift of the parabola if \( h \) is positive and a leftward shift if \( h \) is negative. The \( v \) variable induces an upward shift if positive and a downward shift if negative. Thus, these curves bottom out at the point \((h, v)\). For \( h = 3 \) and \( v = 1 \), we get:

\[ y = (x - 3)^2 + 1 \]

\[ y = (x - 3)(x - 3) + 1 \]

\[ y = x^2 - 3x + 9 + 1 \]

\[ y = x^2 - 6x + 10 \]

Plotting the parabola yields:
We can see from the plot that when $x = 3$, the curve bottoms out and the tangent at the point should be horizontal, having a slope of zero. To the right of that point, the slope rises to the right, so the derivative in that area should be positive. To the left of the bottom point, the slope rises to the left, so the derivative in that area should be negative.

To confirm our intuition, we shall encode the parabola as a sum of terms:

\[
\text{var } y = \text{term}(1,:x,2) \text{ minus term}(6,:x,1) \text{ plus constant}(10);
\]

and take the derivative:

\[
\text{var } y' = y \ . \ \text{diff}(:x);
\]

Let’s look at $y$ and its derivative at $x = 2, 3, 4$:

\[
\begin{align*}
\text{sway} & \triangleright y \ . \ \text{value}(2) \\
\text{INTEGER: 2} \\
\text{sway} & \triangleright y \ . \ \text{value}(3) \\
\text{INTEGER: 1} \\
\text{sway} & \triangleright y \ . \ \text{value}(4) \\
\text{INTEGER: 2} \\
\text{sway} & \triangleright y' \ . \ \text{value}(2) \\
\text{INTEGER: -2} \\
\text{sway} & \triangleright y' \ . \ \text{value}(3) \\
\text{INTEGER: 0} \\
\text{sway} & \triangleright y' \ . \ \text{value}(4) \\
\text{INTEGER: 2}
\end{align*}
\]

Just as we suspected, the slope at the bottom is indeed zero and the slope to the left is negative and the slope to the right is positive.
10.1 Solving problems

Let’s take a break from programming and use the software we’ve developed to solve some problems given in CME.

10.2 Questions
Chapter 11

Peaks and Valleys

According to CME, to find a minima or maxima of a curve with independent variable \( x \), you take the derivative, set the derivative to zero, and then solve for \( x \). At that point, the slope of the original is zero and therefore must be the highest point on the peak or the lowest point in a valley. As CME states, you don’t know for sure if the zero slope represents a peak or if it represents a valley, but it must be one or the other.

The process for solving for \( x \) is much like the process for simplifying an expression; it is a complex task beyond the scope of this text. So we will not be able to find minima/maxima symbolically. However, thanks to Sir Isaac Newton, we will be able to perform this task numerically.

For the rest of this chapter, we will dispense with our object approach and take a functional approach, as the whole purpose for using objects was to perform symbolic tasks. Adapting our previous object approach to perform numeric tasks such as finding minima/maxima is left as an exercise.

11.1 A Blast from the Past

To begin, we must learn how to represent polynomials and their derivatives as numeric Sway functions. Remember our ratio function from Chapters 3 and 4? We used it to compute \( \frac{dy}{dx} \) numerically:

\[
\text{function ratio}(y,x) \\
\quad \{ \\
\quad \quad \text{var dx} = 0.0001; \\
\quad \quad \text{var dy} = y(x + dx) - y(x); \\
\quad \quad \text{dy} / \text{dx}; \\
\quad \}
\]

In Sway, not only are we allowed to return numbers, strings and environments (objects) from functions, we can also return functions themselves. We will use this fact to build numeric derivative functions.

To learn more about returning local functions, see *The Sway Reference Manual*, Chapter 22.

So rather than use the ratio function, which requires us to pass in the same \( y \) function over and over, we will define a function to return a customized ratio that takes only the \( x \) value as an argument. Looking at the code will help immensely:

\[
\text{function derivative}(y) \\
\quad \{ \\
\quad \quad \text{function ratio}(x) \\
\quad \quad \quad \{ \\
\quad \quad \quad \}
\]

65
\begin{verbatim}
var dx = 0.0001;
var dy = y(x + dx) - y(x);
dy / dx;
// ratio;
\}
\}
\end{verbatim}

Note how the function does not call \textit{ratio}, but returns the function bound to the \textit{ratio} variable name instead. Now, we can pass an $x$ value to both the original function $y$ and its derivative.

As always, let’s test, using this polynomial as a guinea pig:

$$3x^2 - 10x + 5$$

Representing this polynomial numerically as a Sway function, we have:

\begin{verbatim}
function y(x)
{
  3 * (x ^ 2) - (10 * x) + 5;
}
var y' = derivative(y);
\end{verbatim}

We would expect the symbolic derivative of $y$ to be:

$$6x - 10$$

Therefore, we would expect $y(2)$ to yield -3 and $y'(2)$ to yield 2.

\begin{verbatim}
sway> y(2);
INTEGER: -3
sway> y'(2);
REAL: 2.0000029988
\end{verbatim}

The imprecision in the second result is again due to working with real numbers rather than integers.\footnote{We could try to increase the precision by using a smaller value for \textit{dx}. However, we run the risk of outstripping the native precision of Sway’s real numbers which carry a limited number of significant digits. Someday, Sway may have \textit{infinite precision} integers and reals, like a good language should.}

\subsection{11.2 A good fixation}

A necessary next step in finding minima/maxima numerically is to be able to find a \textit{fixed point} of a function. If a function has a fixed point (and not all functions do), then there is a value $x$ such that:

$$f(x) = x$$

For example, the cosine function has a fixed point near 0.7390851332:
I say near because Sway does not show all the significant digits, by default.

Finding a fixed point is algorithmically simple:

```splay
function fixedPoint(f, x)
{
    if (f(x) == x)
    {
        x;
    }
    else
    {
        fixedPoint(f, x + f(x) / 2);
    }
}
```

If, for the given \( x \), \( f(x) \neq x \), we try again with a new value of \( x \). Conveniently, the average of \( x \) and \( f(x) \) is a mathematically sound value for our next try.

Although this code illustrates the approach well, there are a few things wrong with it. The first problem is that \( f(x) \) is calculated twice. For efficiency’s sake, we should calculate it once:

```splay
function fixedPoint(f, x)
{
    var next = f(x);
    if (next == x)
    {
        x;
    }
    else
    {
        fixedPoint(f, x + next / 2);
    }
}
```

The next problem is due to the limited precision of real numbers in Sway. Because real numbers cannot be represented exactly, it is usually unwise to make equality comparisons between real numbers at the extremes of their precision. Therefore, unless you know better (and in the case of `fixedPoint`, you don’t know better), you should check if the numbers are separated by some small distance, instead:

```splay
var FIXED_POINT_DELTA = 1e-10;

function fixedPoint(f, x)
{
    var next = f(x);
    if (abs(x - next) < FIXED_POINT_DELTA)
    {
        x;
    }
    else
    {
        fixedPoint(f, x + next / 2);
    }
}
```
The \( \text{abs} \) (absolute value) function is used since \( x - \text{next} \) might be a large negative value, which would also be less than \( 1 \times 10^{-10} \) threshold.\(^2\)

We need an initial value for \( x \) when we call \textit{fixedPoint}. It turns out that 1.0 is almost always a good guess, so let’s hard-wire it. Here is a neat trick for doing that:

\[
\text{function fixedPoint}(f) \\
\{ \\
\quad \text{function helper}(f,x) \\
\quad \{ \\
\quad \quad \text{var next} = f(x); \\
\quad \quad \text{if} \ (\text{abs}(x - \text{next}) < \text{FIXED\_POINT\_DELTA}) \\
\quad \quad \{ \ x; \} \\
\quad \quad \text{else} \\
\quad \quad \{ \ \text{helper}(f, x + \text{next} / 2); \} \\
\quad \} \\
\quad \text{helper}(f,1.0); \\
\}
\]

The essence of the trick is to rename your function for which you wish to fix some arguments to a nice name like \textit{helper}.\(^3\) You then wrap that function with a function having the original name, making \textit{helper} a local function. Then the last thing you do in the wrapper function is call the helper, fixing the values of the desired argument.

We can improve this latest version by noting that \textit{fixedPoint}'s formal parameter \( f \) is bound to the same value as \textit{helper}'s formal parameter \( f \), so we can remove \textit{helper}'s version in both the definition and call:

\[
\text{function fixedPoint}(f) \\
\{ \\
\quad \text{function helper}(x) \\
\quad \{ \\
\quad \quad \text{var next} = f(x); \\
\quad \quad \text{if} \ (\text{abs}(x - \text{next}) < \text{FIXED\_POINT\_DELTA}) \\
\quad \quad \{ \ x; \} \\
\quad \quad \text{else} \\
\quad \quad \{ \ \text{helper}(x + \text{next} / 2); \} \\
\quad \} \\
\quad \text{helper}(1.0); \\
\}
\]

Don’t forget to change the recursive calls as well.

### 11.3 Newton’s Transformer

Newton came up with a clever way to find out where polynomials (and other differentiable functions) have a value of zero. He said that a zero of some function \( f \) is also a fixed point of a new function derived from \( f \). This new function, \( f_{\text{NT}} \), is:

---

\(^2\)Note the attempt to avoid hard-wiring a number like \( 1e-10 \) in the body of \textit{fixedPoint}.

\(^3\)Remember to rename the recursive calls, as well. Forgetting to do so is the most common error in performing this transformation.
Thus, to find a zero of some function, we generate a new function using Newton’s Transform and pass the transformed function to our fixed point finder. The value it returns is our zero!

First, let’s implement the Newton Transform:

```sway
function NewtonTransform(f)
{
    var f' = derivative(f);
    function transform(x)
    {
        x - (f(x) / f'(x));
    }
    transform;
}
```

Now let’s test our polynomial:

\[
y = 3x^2 - 10x + 5
\]

We implement it using a Sway function, as before:

```sway
function y(x)
{
    3 * (x ^ 2) - (10 * x) + 5;
}
```

Where does function \( y \) produce a zero?

```
sway> fixedPoint(NewtonTransform(y));
REAL: 0.6125741133
```

The fixed point function says the fixed point of the transformed \( y \) is at 0.6125741133 (approximately). Let’s feed that result back into \( y \) to see if we get a zero:

```
sway> y(0.6125741133);
REAL: -1.441567e-10
```

At that fixed point, our polynomial has a value of \(-1.441567e-10\), which is very close to zero. Given our inherent imprecision, 0.6125741133 does seem to effectively yield a zero for \( y \).
Chapter 12

Dead Man’s Curve

12.1 Function combination

CME describes a function:

\[ y = \frac{x - 1}{x^2 + 2} \]

for which we wish to find its maximum and minimum. A plot of \( y \) is shown below:

We could write the function \( y \) directly, but suppose you are given access to two functions, \( f \) and \( g \), that represent the numerator and the denominator of \( y \), respectively. Suppose further that you don’t know what they each compute, except by calling them. If you are asked to produce \( y \), what are you to do? You can use function combination!\(^1\)

\[ \text{function combine}(a, \text{op}, b) \]

\(^1\)This technique is very similar to the one used in creating numeric derivative function in the previous chapter.
Here, we create a new function that combines the results of the old functions using the supplied operator.

With \( f \) defined as:

\[
\text{function } f(x) \{ x - 1; \}
\]

and \( g \) defined as:

\[
\text{function } g(x) \{ x ^ 2 + 2; \}
\]

we create \( y \) via the \textit{combine} function:

\[
\text{var } y = \text{combine}(f,/,g);
\]

Now we can find a maxima or minima for \( y \) by finding a zero for \( y' \), using the Newton Transform:

\[
\text{var } y' = \text{derivative}(y);
\]

\[
\text{sway> fixedPoint(}\text{NewtonTransform}(y'));\text{STRING: 2.7320008082}
\]

We now know that \( y \) has a maximum or a minimum around 2.732, but which is it? We can look at the value of the second derivative at the point and it will tell us quite clearly what the answer is. This is why.

If \( y \) has a maximum at that point, the slope to the immediate left is positive, since the curve is rising to its maximum. To the immediate right, the slope is negative, since the curve is now falling away from the maximum. The second derivative tells us how fast and in what direction the slopes are changing, so, in the case of a maximum, the second derivative should be negative. This is because slopes are getting smaller or more negative.

Conversely, if \( y \) has a minimum at that point, then the second derivative will be positive, since the slopes are changing from negative (the curve is descending to the minimum) to positive (the curve is ascending away from the minimum).

For \( y \), we find the second derivative at the zero of \( y' \) to be:

\[
\text{var } y'' = \text{derivative}(y');
\]

\[
\text{sway> } y''(\text{fixedPoint(}\text{NewtonTransform}(y')));\text{STRING: -0.0386717880}
\]

The result is negative, so we have found the maximum of \( y \). If we look at the plot above, we see that our findings make sense; \( y \) appears to have a maximum between \( x = 2 \) and \( x = 3 \). However, there also appears to be a minimum that we have missed, between \( x = 0 \) and \( x = -1 \).
12.2 Other minima and maxima

Why did we find the maximum of $y$ but not its minimum? We can get some insight into this question by looking at the curve of $y'$:

The fixed-point finder in conjunction with the Newton Transform is a curve following system. If the value of $y'$ at the initial point is the same sign as the slope at that point, then the system searches to the left. If they are of different signs, the system searches to the right. Our fixed-point finder is hard-wired to start at 1. We can look at the above plot at $x = 1$ (rightmost blue line) and see that $y'$ at that point is positive, but the slope is negative; the signs are different so the system finds the zero crossing of $y'$ that exists to the right of the starting point. If we set the starting point of the fixed-point finder to $x = -0.5$ (leftmost blue line), $y'$ is still positive, but the slope is also positive. The signs are the same, so the system searches and finds the zero crossing to the left of $x = -0.5$. After modifying our `fixedPoint` function to take a starting point,\(^2\) We see we find a different zero crossing for $y'$:

```
sway> fixedPoint(NewtonTransform(y'),-0.5);
STRING: -0.7321008082
```

This means there is a different minimum or maximum at that point. Which is it? The second derivative can tell us:

```
sway> y''(-0.7321008082)
STRING: 0.5387217894
```

Since the second derivative returns a positive value, we know this is a minima of $y$.

\(^2\)The astute reader will have noticed that we have gotten back to our first attempt at the `fixedPoint` function in the previous chapter. The moral of this story is that, sometimes, you can be too clever.
Chapter 13

Breaking Up is Hard to Do

CME, Chapter XIII, covers two topics: partial fractions and inverse functions. As SPT notes in the last paragraph of the chapter, solving problems of this sort is more of an art than a science. As such, writing programs to split fractions and to find inverse functions become much more difficult.

What we can do, is verify some of the examples in CME to see if what SPT says is indeed true.

13.1 Partial Fractions

The fraction:

\[
\frac{3x + 1}{x^2 - 1}
\]

can be decomposed into the following sum of two partial fractions:

\[
\frac{1}{x + 1} + \frac{2}{x - 1}
\]

We can represent the undecomposed fraction as a calculus object:

```sway
var u = (term(3,:x,1) plus 1) div (term(1,:x,2) minus 1);
```

Likewise, we can represent the decomposed fraction as:

```sway
var d = 1 div (:x plus 1) plus (2 div (:x minus 1));
```

Note that the above constructions assumes the toCalculus enhancements \(^1\) for the automatic conversion of symbols and numbers to calculus variables and constants for plus and minus are in effect.

Let’s test to see if \(u\) and \(d\) are equivalent: First \(u\):

```sway
sway> u . toString();
STRING: (3x + 1) / (x^2 - 1)
```

\(^1\)See chapter 9.
Now, $d$:

\[
\text{sway> } d . \text{toString();} \\
\text{STRING: } \frac{1}{(x + 1)} + \frac{2}{(x - 1)}
\]

\[
\text{sway> } d . \text{value(0);} \\
\text{STRING: } -1.0000000000
\]

\[
\text{sway> } d . \text{value(13);} \\
\text{STRING: } 0.2380952381
\]

We only tested two values, but, given the fact that SPT says the two fractions are equivalent, I don’t think we need to test further.

What about their derivatives? If $u$ and $d$ are equivalent, then their derivatives should be equivalent as well:

\[
\text{var } u' = u . \text{diff(:x);} \\
\text{var } d' = d . \text{diff(:x);} \\
\text{sway> } u' . \text{toString();} \\
\text{STRING: } \frac{(3x + 1) \cdot 2x - (x^2 - 1) \cdot 3}{(x^2 - 1) \cdot (x^2 - 1)}
\]

\[
\text{sway> } d' . \text{toString();} \\
\text{STRING: } \frac{1}{((x + 1) \cdot (x + 1))} + \frac{2}{((x - 1) \cdot (x - 1))}
\]

Hmmmm. They don’t look equivalent. We can start to ascertain that they are by trying a couple of values:

\[
\text{sway> } u' . \text{value(0);} \\
\text{STRING: } 3.0000000000
\]

\[
\text{sway> } u' . \text{value(13);} \\
\text{STRING: } 0.0189909297
\]

\[
\text{sway> } d' . \text{value(0);} \\
\text{STRING: } 3.0000000000
\]

\[
\text{sway> } d' . \text{value(13);} \\
\text{STRING: } 0.0189909297
\]

Again, we can’t say for sure the derivatives are the same (using our testing strategy, we would need to test an infinite number of values to say for sure), but it certainly appears that they are.

13.2 Inverse Functions

Like the partial fractions, we can investigate the following fact: the derivative of a function times the derivative of the inverse of the function is a constant one. Here’s a function and its inverse:

\[
y = 4x^2
\]
13.2. **INVERSE FUNCTIONS**

\[ x = \frac{y^\frac{1}{2}}{2} \]

```sway
sway> y . value(x . value(13));
STRING: 13.000000000
sway> x . value(y . value(13));
STRING: 13.000000000

sway> y . value(x . value(101));
STRING: 101.00000000
sway> x . value(y . value(101));
STRING: 101.00000000
```

These functions do seem to be inverses of each other.

If two functions are inverses, then their derivatives are reciprocals. We can take advantage of this fact to test our calculus system. If we invert a function, take the inverse’s derivative, do a nifty substitution, then take the reciprocal of the result, we end up with an alternate derivative of the original function. If our system is working correctly, we should get

First, we’ll go through the steps mathematically. We start with the inverted function:

\[ x = \frac{y^\frac{1}{2}}{2} \]

and differentiate it:

\[ x' = \frac{1}{4y^2} \]

Next we substitute the original value of \( y \) back into \( x' \).

\[ x' = \frac{1}{4[4x^2]^{\frac{1}{2}}} \]

Simplifying yields:

\[ x' = \frac{1}{4[4^\frac{1}{2}(x^2)^{\frac{1}{2}}]} \]

\[ x' = \frac{1}{4[2x]} \]

\[ x' = \frac{1}{8x} \]

Since \( y \) is:

\[ y = 4x^2 \]
the derivative of $y$ is:

$$y' = 8x$$

Indeed, the derivatives are reciprocals.

Now, let’s do the same process, this time programatically. And while we are at it, let’s wrap up all the steps into a useful function:

```javascript
function alt-derivative(original,inverse,wrtv) {
  // differentiate the inverse function
  var x' = inverseFunction . diff(wrtv);
  // substitute the original function
  x' . iv = original;
  // return the reciprocal
  1 div x';
}
```

```javascript
var y' = y . diff(:x);
var x' = x . diff(:y);
var alt-y';

x' . iv = y;

sway> y' . value(2);
REAL: 16.00000000

sway> alt-y' . value(y);
REAL: 16.00000000
```
Chapter 14

The Rich get Richer
Chapter 15

Auld Lang Sine

You have, in all likelihood, previously made acquaintance with sines and cosines. Sinusoids (which include sines and cosines) are periodic functions. Periodic functions have the following property:

\[ f(x) = f(x + p) = f(x + 2p) = \ldots \]

for all \( x \) and certain values of \( p \). The smallest value of \( p \) for which the above equation holds is known as the period of the function.

If you not familiar with sines and cosine functions, this is what a sine wave looks like:

Note that for sine waves without phase and frequency shifts (as above), when \( x \) is zero, the amplitude (\( y \)-value) of the sine wave is zero (with a rising slope). The wave reaches zero again (with a rising slope) \( x = 2\pi \). Contrast this with the cosine wave:
Unlike the sine wave, which has zero-crossings at multiples of π, the cosine wave has peaks and troughs at multiples of π. As with the sine wave, the period, or peak to peak length, is 2π.

15.1 The general sine wave

If you look closely at the two waves, you’ll see that the cosine is just the sine wave shifted $\frac{\pi}{2}$ units to the left. So, we can define cosine in terms of sine:

$$\cos(x) = \sin(x + \frac{\pi}{2})$$

Note that mathematicians often abbreviate sine to sin and cosine to cos. The above illustrates one of the common ways to modify a sine wave: shifting the phase of the wave, for which the symbol $\theta$ is often used:

$$y = \sin(x + \theta);$$

Another modification is to change the amplitude of a sine wave or, on other words, to change how high and how low the peaks range. The variable $a$ is used to indicate amplitude, so the formula for a sine wave that allows amplitude modification is:

$$y = a \sin(x + \theta);$$

Finally, it is common to change the frequency of sine waves, or how many peaks (or troughs) are in a given region. The variable $\omega$ is often used for this task:

$$y = a \sin(\omega x + \theta);$$

If we set $\omega$ to 2, we will get twice as many peaks (or troughs) within a given region. The inverse of frequency is period, so setting $\omega$ to 2 will shorten the peak to peak distance by half. In the particular case of frequency 2, the period of the sine wave would be $\pi$. 
Here's a neat trick. If you want to find the location of the 'first' zero crossing of a general sign wave of this form, change the sign of both the phase shift and the operator combining it with \( x \) and then factor out \( \omega \).

For cosine, with unit amplitude and frequency doubled, we have:

\[
\cos(x) = \sin(2x + \frac{\pi}{2})
\]

\[
\cos(x) = \sin(2x - \left(-\frac{\pi}{2}\right))
\]

\[
\cos(x) = \sin(2[x - \left(-\frac{\pi}{2}\right)])
\]

\[
\cos(x) = \sin(2[x - \left(-\frac{\pi}{4}\right)])
\]

This version of cosine has its first zero crossing at \( x = -\frac{\pi}{4} \). In other words, to find the first zero crossing, divide the phase shift by the frequency and negate it (assuming that the phase shift is being added in).

15.2 Implementing the general sine wave

Sway has the functions \( \sin \) and \( \cos \) built-in, but these functions assume amplitude = 1, frequency = 1, and phase shift = 0. To implement \( \sin \) and \( \cos \), we will take our usual object approach:

```javascript
function sine(amp,freq,shift)
{
    function value(x)
    {
        amp * sin(freq * x + shift);
    }
    this;
}

function cosine(amp,freq,shift)
{
    sine(amp,freq,shift + (pi() / 2));
}
```

Note how we make cosine a wrapper for sine as cosine is just sine with a phase shift.

Let's take that little trick we learned in the previous section for finding the 'first' zero crossing and implement it:

```javascript
function sine(amp,freq,shift)
{

```

---

1. Contrast this with the cosine wave with unit amplitude and frequency (shown at the beginning of this chapter), which has a zero crossing at \( x = -\frac{\pi}{4} \).

2. There is also a generalized cosine function: \( y = a\cos(\omega x + \theta) \). Of course, this is equal to \( a\sin(\omega x + \frac{\pi}{2} + \theta) \).
function value(x) { ... }
function firstZero()
{
    // phase shift is added in so just divide and negate
    -(real(shift) / freq);
}
this;
}

Let’s see if it works for cosine with frequency 2:

```
var w = cosine(1,2,0);
```

```
sway> -(pi() / 4);
REAL: -0.7853981634
```

```
sway> w . firstZero();
REAL: -0.7853981634
```

Good. Let’s also check that the value of the sine wave is indeed zero at that point:

```
sway> var fz = w . firstZero();
REAL: -0.7853981634
```

```
sway> w . value(fz);
w . value(fz) is 0.000000e+00
```

Bingo!

### 15.3 Differentiating sine and cosine

As SPT points out in Chapter XV in CMT, the derivative of sine is cosine and the derivative of cosine is the negative of sine.

Using the first rule, we can add a `diff` function to the `sine` constructor. Of course, in keeping with our unassuming differentiation system, we need to pass in independent and with-respect-to variables, as appropriate:

```
function sine(amp,freq,shift)
{
    function value(x) { ... }
    function firstZero() { ... }
    function diff()
    {
        cosine(amp,freq,shift);
    }
    this;
}
```

Of course, this implementation assumes that the independent variable and the with-respect-to variable are one and the same. It also assumes that the independent variable is just a symbol. As such, we could not construct a sine wave of the form;
\[ y = 3 \sin(2x^2 + \pi) \]

To do so, we will have to take the same approach as for terms, by allowing the independent variable to be a differentiable object. Recall the term constructor:

```javascript
function term(a, iv, n)
{
    function value(x) { ... }
    function toString() { ... }
    function diff(wrtv)
    {
        if (n == 0)
        {
            constant(0);
        }
        else
        {
            term(a * n, iv, n - 1) times iv . diff(wrtv);
        }
    }
}
```

Recall also how the `diff` function implements the chain rule. We will need to follow the same strategy for our `sine` constructor:

```javascript
function sine(amp, freq, iv, shift)
{
    function value(x) { ... }
    function firstZero() { ... }
    function diff(wrtv)
    {
        cosine(amp, freq, iv, shift) times iv . diff(wrtv);
    }
}
```

All that’s left for us to do is implement our visualization for `sine` (and, of course, test):

```javascript
function toString()
{
    "" + amp +
    " sin(" + freq +
    "(" + iv . toString() + ")" + shift + ")";
}
```
15.4 What is -sin?

According to CMT, the derivative of sine is cosine and the derivative of cosine is -sin. Therefore, the second derivative of sine is -sine.

15.5 Questions

1. What is the difference between the function $\sin(x + \frac{\pi}{2})$ and the function $\sin(x) + \frac{\pi}{2}$?

2. Why shouldn’t we name $sine$ and $cosine$ constructors $\sin$ and $\cos$?

3. Add the following simplification to the $sine$ constructor. If the phase shift is equal to or greater than $2\pi$, subtract off $2\pi$.

4. Explain why the previous simplification is mathematically valid.

5. Simplify the construction of sine objects so that a zero object is generated if the amplitude is zero.

6. Simplify the construction of sine objects so that a constant term object is generated if the frequency is zero.

7. Simplify the visualization of sine objects so that the amplitude, frequency, and phase shift are omitted if they are 1, 1, and 0, respectively.

8. Find and add other visualization mods. Hint: what should your visualization look like if the frequency is $\neq 1$ but the independent variable is a term with a coefficient $\neq 1$?
Chapter 16

Do What You Can
Chapter 17

And Now For Something Completely Different

Enough with the differentiation, already! Time for something new. New, but not all that different. Once you have learned about differentiation in your Math classes, you will surely start to learn about integration. As stated very early on, integration is the summing up of all the tiny (infinitesimal) pieces of a curve. So if \( dy \) represents one of the (infinitely) many bits of the curve \( y \), then:

\[
\int y \, dy = y
\]

Indeed, if you sum up all the bits of a thing, you get the thing itself.\(^1\)

17.1 Summing a series

As with differentiation, there are two kinds of integration, numeric and symbolic. Like differentiation, numeric integration gives an approximate result while symbolic integration gives an exact result. Since numeric integration involves the actual summing up of a bunch of little bits, let’s get some practice summing.

What is the sum of:

\[
1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots
\]

with an infinite number of fractions following the pattern. We have a two issues to solve before we can write code to give us the answer. The first is that the sum seems to be composed of two different things, a whole number (1) and a bunch of fractions.

Life is so much easier when all things are the same; if this is so, we have no need of cluttering up our code with \( if \) expressions to decide on what kind of thing we are working on. \(^2\) In general, if we have two kinds of things we wish to treat the same, we need to either make the first look like the second, the second look like the first, or both look like some other thing. In the specific case of this summation, we need to either:

- make the fractions look like whole numbers,
- make the whole number look like a fraction, or
- make the whole number and the fractions look like a third kind of thing

\(^1\) Assuming you don’t believe in that magical marketing term, synergy.

\(^2\) We saw this previously when we made a constant look like a term.
Of the three strategies, the second seems to be the easiest route since a whole number is easily represented as a fraction by placing a one in the denominator:

\[
\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots
\]

To learn more about iterative loops, see *The Sway Reference Manual*, Chapter 12.

The remaining issue we need to solve is how to structure our code. It is clear that we will need some kind of loop, either recursive or iterative, to add up as many fractions as desired. In order to use a loop, we usually need to make the items we are looping over look the same or at least be a function of the loop counter. In our case, how do we make each fraction look alike?

If we look at the denominator, we see that each can be described as a power of two:

\[
\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \frac{1}{2^4} + \ldots
\]

It seems that we can use a loop counter (starting at zero) as the exponent in the denominator of each fraction. With these two issues out of the way, we can begin to write some code. Let’s try the recursive route first.

### 17.1.1 The recursive way

We will use \(n\) as our loop counter:

```javascript
function total(n)
{
  1 / (2 ^ n) + total(n + 1);
}
```

If we call `total` with an initial value of zero for \(n\), we should get:

\[
1 / (2 ^ 0) + total(1)
\]

Upon the next recursive call, we should get:

\[
1 / (2 ^ 0) + [1 / (2 ^ 1) + total(2)]
\]

And then:

\[
1 / (2 ^ 0) + [1 / (2 ^ 1) + [1 / (2 ^ 2) + total(3)]]
\]

This is exactly what we want, but we are going to end up summing an infinite number of terms. We have neglected to have a base case for stopping the recursion. Let’s pass in the number of fractions we wish to sum up as `max`:
function total(n,max)
{
    if (n == max)
        { 0; }
    else
        { 1 / (2 ^ n) + total(n + 1,max); }
}

Note that we are using the function call syntax for if in order to shorten the length of our function. The recursion stops when \( n \) reaches \( max \) with zero being returned in that case.

What do we get if we total up five fractions?

sway> total(0,5);
REAL: 1.9375000000

Is this correct? Let’s add up five fractions explicitly:

sway> 1.0 / 1 + (1.0 / 2) + (1.0 / 4) + (1.0 / 8) + (1.0 / 16);
REAL: 1.9375000000

Seems to be. How about ten fractions?

sway> total(0,10);
REAL: 1.9980468750

Fifty fractions?

sway> total(0,50);
REAL: 2.0000000000

One hundred fractions?

sway> total(0,100);
REAL: 2.0000000000

These results lead us to believe that if we were to add up these fractions out to infinity, we would get 2 as a total. Of course, since our result is a real number, we need to be wary of trusting it absolutely, but in this case, I’d be willing to bet the result is correct.

How about we add some visualization to this function? We probably should have done this first to make sure our code was behaving as expected:

function total(n,max)
{
    if (n == max)
        { 0; }
    else
        { 1 / (2 ^ n) + total(n + 1,max); }
CHAPTER 17. AND NOW FOR SOMETHING COMPLETELY DIFFERENT

```
println("0");
0;
}
else
{
    var denom = 2 ^ n;
    print("1/",integer(denom)," + ");
    1 / denom + total(n + 1,max);
}
```

Note how we 'precomputed' the denominator since we need it twice, once for the visualization and one for the actual computation. We also use the `integer` function to convert the real number that exponentiation produces back to an integer.

```
sway>total(0,5);
1/1 + 1/2 + 1/4 + 1/8 + 1/16 + 0
REAL: 1.9375000000
```

Looking good! Of course, our visualization does not produce a valid Sway expression (whitespace and precedence problems), but that is rarely necessary for a visualization.

### 17.1.2 An iterative approach

Let’s try using an iterative loop instead. There is a standard methodology for converting a recursive loop to an iterative loop. The first step is to initialize a local variable, say `result`, to the recursive loop’s base case return value. In this case, the return value is zero:

```
function total(n,max)
{
    var result = 0;

    return result;
}
```

Now we add a while loop with the opposite of the test found in the recursive loop’s `if` expression:

```
function total(n,max)
{
    var result = 0;

    while (not(n == max)) //better is (n != max)
    {
    }

    return result;
}
```

In the body of the loop, we place the recursive case calculation:
17.1. SUMMING A SERIES

```javascript
function total(n,max)
{
    var result = 0;

    while (n != max))
    {
        var denom = 2 ^ n;
        1 / denom + total(n + 1,max);
    }

    return result;
}
```

replacing the recursive call with `result`:

```javascript
function total(n,max)
{
    var result = 0;

    while (n != max))
    {
        var denom = 2 ^ n;
        1 / denom + result;  //was total(n + 1,max)
    }

    return result;
}
```

and assigning the whole expression back to `result`:

```javascript
function total(n,max)
{
    var result = 0;

    while (n != max))
    {
        var denom = 2 ^ n;
        result = 1 / denom + result;  //was total(n + 1,max)
    }

    return result;
}
```

Finally, if any variable was updated in the original recursive call, update the variable at the bottom of the loop using assignment:

```javascript
function total(n,max)
{
    var result = 0;

    while (n != max))
    {
```
Voila! We’re done. This technique generally works well and will even work if there are multiple base cases and multiple recursive cases. Sometimes, however, things go a little bit wrong in the transformation. Here’s an example: let’s convert the recursive version of the greatest common divisor function:

```javascript
function gcd(n, d) {
    if (d == 0) {
        return n;
    }
    else {
        gcd(d, n % d);
    }
}
```

We start with the base case return value:

```javascript
function gcd(n, d) {
    var result = n;
    return result;
}
```

Next, we add a while loop:

```javascript
function gcd(n, d) {
    var result = n;
    while (d != 0) {
        }
    return result;
}
```

Then we copy over the recursive case calculation:

```javascript
function gcd(n, d) {
    var result = n;
    while (d != 0) {
        }
    return result;
}
```
We replace the recursive call in the calculation by \textit{result}:

\begin{verbatim}
  function gcd(n,d)
  {
      var result = n;
      while (d != 0)
      {
          result;  //was gcd(d,n % d);
          n = d;
          d = n % d;
      }
      return result;
  }
\end{verbatim}

Now we update the variables that change in the recursive call:

\begin{verbatim}
  function gcd(n,d)
  {
      var result = n;
      while (d != 0)
      {
          result;  //was gcd(d,n % d);
          n = d;
          d = n % d;
      }
      return result;
  }
\end{verbatim}

We should be done at this point, but we still have a couple of problems. The first is that the statement:

\begin{verbatim}
  result;
\end{verbatim}

in the body of the while loop isn’t doing anything useful. This is our first clue that something is amiss. For now, let’s delete it:

\begin{verbatim}
  function gcd(n,d)
  {
      var result = n;
      while (d != 0)
      {
          n = d;
          d = n % d;
      }
      return result;
  }
\end{verbatim}
The second problem is that in the recursive call, the update to variable $d$ used the old value of $n$ but in our transformation, the update to $d$ uses the new value of $n$. We need to save the old value and we use a temporary variable to do so:

```javascript
function gcd(n, d)
{
    var result = n;
    while (d != 0)
    {
        var temp = n;
        n = d;
        d = temp % d;
    }
    return result;
}
```

The final problem is that result never changes. That’s because we initially set result to the original value of $n$, not the last value of $n$. Note that the recursive version uses the last value of $n$ as desired. Thus, we update `result` after the while loop terminates to get the last value of $n$:

```javascript
function gcd(n, d)
{
    var result = n;
    while (d != 0)
    {
        var temp = n;
        n = d;
        d = temp % d;
    }
    result = n;
    return result;
}
```

Of course, we can clean up our function up a bit by realizing that `result` isn’t doing anything at all; we can just return the last value of $n$:

```javascript
function gcd(n, d)
{
    n = d;
    d = n % d;
}
return result;
}
```
{ }
while (d != 0)
{
    var temp = n;
    n = d;
    d = temp % d;
}
return n;
}

This technique, at its best, gets you the right answer immediately. At worst, it gets you well on the way towards the right answer.

## 17.2 Summing as an abstraction

Note that in both our recursive and iterative solutions to the original summation, we have hard-wired the function that computes the next element in the sequence (namely the inverted exponentiation). An important design strategy for writing computer programs is called *separation of concerns*. With this strategy, we try to write functions, or sets of functions, so that each function performs a single task. In our solutions, both the generation of the next fraction (one concern) and the summing of those fractions (another concern), are found in a single function. We can separate those concerns into individual functions. One function generates the fractions to be summed; the other performs the actual summing, given the previously generated collection of fractions. Our first task, then, is to generate the collection of fractions. We will examine to ways of gathering a collection together, *arrays* and *lists*.

To learn more about arrays and lists, see *The Sway Reference Manual*, Chapter 15.

Here is the code for generating a list of the desired fractions:

```
function invPowOfTwo(n,max)
{
    if (n == max)
    { :null; }
    else
    { 1 / (2 ^ n) join invPowOfTwo(n + 1,max); }
}
```

If I wanted an array instead of a list, I might write the function as:

```
function invPowOfTwo(n,max)
{
    var a = allocate(max);

    while (n < max)
    {
        a[n] = 1 / (2 ^ n);
        n = n + 1;
    }
}
```

---

3 Lists lend themselves to recursive solutions, arrays to iterative solutions.
Either way, once I have my collection of fractions, I can now sum them with a general purpose summer:

```javascript
function sum(items)
{
    if (items == :null)
    { 0; }
    else
    { head(items) + sum(tail(items)); }
}
```

Whether I pass an array or list to \textit{sum}, I get the total of all the items in the collection.\footnote{Don’t try this with other languages! In almost every other language, the operators/functions for decomposing a list are different than decomposing an array.}

I could also have written \textit{sum} iteratively:

```javascript
include("basics");

function sum(items)
{
    var i;
    var total = 0;

    for-each(i,items)
    {
        total = total + i;
    }

    total;
}
```

This version will also sum up the collection of items in either list or array form, but for one of these forms, the process is rather slow (see the exercises).

The advantage of separating the generation of elements from the process of summing is this. Once we have our list of elements, we can do more things than just sum with them. We can visualize them, invert them, (after inverting) find the ones that are Mersenne primes plus 1, and so on. Likewise, we can use our summing function to sum up other kinds of collections.

### 17.2.1 Pseudo-infinite sequences

What if we go through the trouble a generating a set of elements to be summed or processed in some fashion and neglected to produce a large enough collection? We are stuck with regenerating the collection again, throwing away all the work we did previously.

The problem is we had to specify the max before we perform the summing:

```javascript
function invPowOfTwo(n,max)
```
It sure would be nice if we didn’t have to specify in advance how large the collection should be. Not requiring a `max` also would simplify the code:

```sly
function invPowOfTwo(n)
{
  1 / (2 ^ n) join invPowOfTwo(n + 1);
}
```

The problem is, without a base case, we will fall into an infinite recursive loop. However, we can avoid this pitfall through delayed or lazy evaluation.

To learn more about lazy evaluation, see *The Sway Reference Manual*, Chapter 19.

Let’s write a version of `invPowOfTwo` that delays evaluation of the recursive call:

```sly
function invPowOfTwo(n)
{
  1 / (2 ^ n) join delay(invPowOfTwo(n + 1));
}
```

Note the use of the `delay` function wrapped around the recursive call. What `delay` does is save everything needed to compute the recursive call without actually making the call. The actual call can be made later using the information saved by `delay`.

Let’s look at the result of calling our new function:

```sly
sway> var items = invPowOfTwo(0);
LIST: (1.0000000000 # <THUNK 11167>)
```

We see the first element of our list is 1.0, as expected. Rather than seeing the rest of our list, however, we see that a `thunk` has been glued on as the tail of the list (the sharp sign signifies that the tail of the list is not a proper list). We can examine the thunk:

```sly
sway> var t = tail(items);
THUNK: <THUNK 11167>

sway> pp(t);
<THUNK 11086>:
  context: <OBJECT 11071>
  code: invPowOfTwo(n + 1)
THUNK: <THUNK 11086>
```
We see that the code field of the thunk object holds the actual call. The context field holds an environment containing the values of \( n \) and \( \text{invPowOfTwo} \), all that is needed to make the actual call.

A list of this sort is known as a delayed stream or stream for short. The stream metaphor is used since as long as there is source of water (memory), the stream will never run dry.

How do we get the elements other than the first element out of the stream? With the force function. The force function, when given a thunk as an argument, performs the evaluation that was delayed in the first place:

```
sway> head(items);
REAL: 1.0000000000

sway> tail(items);
THUNK: <THUNK 11167>

sway> force(tail(items));
LIST: (0.5000000000 # <THUNK 11193>)
```

By forcing the tail of the original list, we end up with a new list generated by the call \( \text{invPowOfTwo}(n + 1) \). When this call was delayed, \( n \) had a value of zero. Now that the call is being made, \( \text{invPowOfTwo} \) is passed a value of one. This generates a new list with \( \frac{1}{2} \) at the head and another delayed recursive call. This time, delay will remember that \( n \) has a value of one instead of zero.

Let’s define a function to obtain any element of a stream. We will pass in the index and then successively force the tail of the list until we get to the correct element:

```javascript
function streamIndex(s,i)
{
    if (index == 0) // the head of the stream is desired
        { head(s); }
    else
        { streamIndex(force(tail(s)),i - 1); }
}
```

Note that if we want element \( i \) of a list with more than \( i \) elements, that is the same as wanting element \( i - 1 \) of the tail of the list. When we take the tail of the list, the desired element appears to move one step closer to the front of the list.\(^5\)

Now we can sum a stream:

```javascript
function sumStream(s,count)
{
    if (count == 0)
        { 0; }
    else
        { head(s) + sumStream(force(tail(s)),count - 1); }
}
```

without having to worry about how many elements were originally generated. Iteratively, summing a stream might look like:

\(^5\)This is very important. Convince yourself this is so.
function sumStream(s, count)
{
    var total = 0;
    while (count > 0)
    {
        total = total + head(s);
        s = force(tail(s));
        count = count + 1;
    }
    total;
}

17.2.2 Streams, the easy way

There is a simpler way to generate streams than using the delay function. You can automatically delay the evaluation of any argument to a function by naming its corresponding formal parameter with a name starting with a dollar sign.

We start by defining a function whose purpose is to join two items together but delaying the evaluation of the second item:

    function ##(a,$b)
    {
        a join $b;
    }

Since the second formal parameter of the ## function begins with a dollar sign, the second argument in a call to ## will be delayed. Now we rewrite invPowOfTwo:

    function invPowOfTwoStream(n)
    {
        1 / (2 ^ n) ## invPowOfTwoStream(n + 1);
    }

Once the stream is made, we can define an analog to the tail function; in this new function, we force the tail:

    function #(stream)
    {
        force(tail(stream));
    }

Now, we rewrite sumStream to show off our new tail-like function:

    function sumStream(s, count)
    {
        if (count == 0)
            { 0; }
With the \#\# and \# functions, we now have a convenient and elegant way to create and manipulate streams.

A useful companion function extracts the $i^{th}$ value from a stream $s$:

\[
\text{function sref(s,i) \{} \\
\quad \text{if (i == 0, head(s), sref(#(s), i - 1));} \\
\}\]

If index $i$ is zero, we return the head of the stream. If not, we shorten the stream and reduce the index by one. We can now look at individual items in the stream:

\[
\text{var s = invPowOfTwoStream(0);} \\
\text{sway> sref(s,0); REAL: 1.0000000000} \\
\text{sway> sref(s,1); REAL: 0.5000000000} \\
\text{sway> sref(s,2); REAL: 0.2500000000} \\
\text{sway> sref(s,3); REAL: 0.1250000000} \\
\text{sway> sref(s,4); REAL: 0.0625000000} \\
\]

The great thing about streams is we never compute anything until we actually need it.

### 17.3 Numeric Integration

Now that we have a nice set of tools for performing summations, we can now move on to integration. Suppose we wish to numerically integrate the function

\[
y = x^2
\]

Geometrically speaking, integration is finding the area under a curve. Suppose we wish to find the area under the curve $y = x^2$ between $x = 1$ and $x = 4$. In other words, we wish to find the area of the hashed region in the figure below:
17.3. NUMERIC INTEGRATION

The common way to solve this problem is to use the so-called trapezoidal rule. Instead of finding the area under the curve, we find that area of a trapezoid that approximates the area under the curve. Specifically, if we wish to find the area under the curve \( f(x) \) between \( x = a \) and \( x = b \), we find instead the area of the quadrilateral whose sides are:

\[
(a, 0), (a, f(a))
\]

\[
(a, f(a)), (b, f(b))
\]

\[
(b, f(b)), (b, 0)
\]

\[
(b, 0), (a, 0)
\]

For our parabola \( y = x^2 \), the trapezoid generated by \( x = 1 \) and \( x = 4 \) is:
We can define a function to calculate the area of this trapezoid:

```javascript
function trapArea(base, left, right) {
    var baseRectangle = base * left;
    var topTriangle = right - left * base / 2.0;
    baseRectangle + topTriangle;
}
```

The area of a trapezoid is computed by dividing the trapezoid into two pieces, a rectangular base upon which sits a triangular top.

For our example, the base is 3 units long, the left side is 1 unit long, and the right side is 16 units long.

```javascript
sway> trapArea(3,1,16);
REAL: 25.500000000
```

So, our estimate of the area under curve is 25.5 square units. We know this to be an over-estimate because, in the figure, the trapezoid is bigger than the area we desire.

We can get a better approximation by dividing up our one large trapezoid into many small ones:
In the figure, we have divided up the large trapezoid into six trapezoids, each having a base of 0.5 units wide. Now we calculate the areas of each trapezoid and sum them together:

\[
\text{sway}\triangleright \text{trapArea}(0.5,1.0,2.25) \\
\text{more}\triangleright + \text{trapArea}(0.5,2.25,4.0) \\
\text{more}\triangleright + \text{trapArea}(0.5,4.0,6.25) \\
\text{more}\triangleright + \text{trapArea}(0.5,6.25,9.0) \\
\text{more}\triangleright + \text{trapArea}(0.5,9.0,12.25) \\
\text{more}\triangleright + \text{trapArea}(0.5,12.25,16.0) \\
\text{REAL:} \ 21.125000000
\]

There are a few things to note about this result. Being a Computer Scientist, I did not actually calculate and type the above into Sway by hand, but wrote a function to generate what looks like an interaction with Sway. The second, is that, from the figures above, 21.125 is a closer approximation of the desired area than the 25.5 result we got with a single trapezoid. This is not an optical illusion, either. The precise value of the area under the curve is 21 square units. In fact, as we divide up the area into ever narrower trapezoids, our approximation will have less and less error. When our trapezoids, are infinitely narrow, our approximation will be exact.

The function I wrote to generate the simulated interaction with Sway is the basis for the following function that generates a list of trapezoid areas that can be used to perform a numerical integration for any curve \[y = f(x)\]:

```swoyer
function trapezoids(f,a,b,step)
{
    if (a >= b)
        { :null; }
    else
        {
            var base = step;
            ...
        }
}
```

\(^6\text{We will not be able to verify this until we learn how to integrate symbolically.}\)
Along with our general purpose summing function, we can now see that as our trapezoids become narrower and narrower, our approximation becomes better and better:

```sway
class trapezoid extends event {
  var base;
  trapezoid( f, a, b, step ) := {
    this.base = b - a;
    var left = f(a);
    var right = f(a + step);
    trapArea(base, left, right) join trapezoids(f, a + step, b, step);
  }
}
```

Oops, well our approximations were getting better except for the last one. What went wrong? Again, it is the imprecision of real numbers that tripped us up. Looking closely at the integrate function, we see that we stop computing trapezoids when \( a \geq b \). When \( a \) was 4.0 and \( b \) was 4, the computation should have stopped. But \( a \) actually was \( 3.9999999999 \), instead of 4.0, so the area of one extra trapezoid was added. The test should have been:

\[
a >= b - (step / 2.0)
\]

We stop when \( a \) is within half a step of \( b \). Now, our trapezoids function works much, much better:

```sway
sway> sum(trapezoids(y,1,4,3.0));
REAL Numero: 25.5000000000
sway> sum(trapezoids(y,1,4,0.5));
REAL Numero: 21.1250000000
sway> sum(trapezoids(y,1,4,0.1));
REAL Numero: 21.0050000000
sway> sum(trapezoids(y,1,4,0.01));
REAL Numero: 21.1604505000
```

We can clearly see the approximated area approaching the desired value of 21 square units.

### 17.4 The power of streams

We can use the power of streams to generate ever closer approximations to the desired area:

```sway
function areas(f,a,b,step)
{
  sum(trapezoids(f,a,b,step)) ## areas(f,a,b,step / 2.0);
}
```

We start by creating a stream of areas. The first area has a single trapezoid since the step (or base) is three units long.
17.5. QUESTIONS

\[
\text{var a = areas(y,1,4,3);}
\]

If we look at the first computed area in the stream, we see the result we expect for a single trapezoid:

\[
\text{sway> sref(a,0);}
\]
\[
\text{REAL: 25.500000000}
\]

The second area has two trapezoids, since the areas function divides the step in half every time a new area is added to the stream:

\[
\text{sway> sref(a,1);}
\]
\[
\text{REAL: 22.125000000}
\]

Subsequent areas in the stream become closer to the desired 21 square units:

\[
\text{sway> sref(a,2);}
\]
\[
\text{REAL: 21.281250000}
\]
\[
\text{sway> sref(a,4);}
\]
\[
\text{REAL: 21.017578125}
\]
\[
\text{sway> sref(a,8);}
\]
\[
\text{REAL: 21.000068665}
\]

As before, we don’t calculate an area

17.5 Questions

1. Does the trapezoid rule always over-estimate? Explain your answer.

2. Look up Horner’s rule and define a function named horner. Your horner function should take the same arguments as the integrate function and in the same order. Of course, it should also produce the same result.

3. In truth, our streams are not very efficient. Suppose, we ask for the \(n\)th item in a stream. We calculate the value of every item up to and including the desired item. We cannot expect to do better than this, but if we subsequently ask for the \(n + 1\) item, we recalculate all the preceding items again. Modify the \# function to save previously calculated values. You will need to assign to the tail of the given stream. For example, to assign the value of \(z\) to the tail of the stream \(s\), you would use the expression:

\[
\text{s tail= z;}
\]

4. Why does the iterative summing procedure much slower for large lists than large arrays?