# SICP Exercise Solutions for Section 1.3

Paul L. Snyder

April 3, 2014

## Contents

**1 Section 1.3.1** **3**
- 1.1 Exercise 1.29: Implementing Simpson's Rule . . . 3
  - 1.1.1 Problem . . . 3
  - 1.1.2 Answer . . . 3
- 1.2 Exercise 1.30: Iterative `sum` . . . 7
  - 1.2.1 Problem . . . 7
  - 1.2.2 Answer . . . 7
- 1.3 Exercise 1.31: A `product` procedure . . . 8
  - 1.3.1 Problem . . . 8
  - 1.3.2 Answer . . . 8
- 1.4 Exercise 1.32: Abstracting `accumulate` . . . 10
  - 1.4.1 Problem . . . 10
  - 1.4.2 Answer . . . 10
- 1.5 Exercise 1.33: Further generalization of accumulation . . . 11
  - 1.5.1 Problem . . . 11
  - 1.5.2 Answer . . . 12

**2 Section 1.3.2** **13**
- 2.1 Exercise 1.34: Perverse self-application . . . 13
  - 2.1.1 Problem . . . 13
  - 2.1.2 Answer . . . 14

**3 Section 1.3.3** **14**
- 3.1 **TODO** Exercise 1.35: The fixed point $\phi$ . . . 14
  - 3.1.1 Problem . . . 14
  - 3.1.2 Answer . . . 14
- 3.2 Exercise 1.36: Observing `fixed-point` approximations . . . 14
  - 3.2.1 Problem . . . 14

3.2.2 Answer . . . 15
3.3 Exercise 1.37: Infinite continued fractions . . . 17
3.3.1 Problem . . . 17
3.3.2 Answer . . . 18
3.4 Exercise 1.38: Euler's continued fraction for $e - 2$ . . . 20
3.4.1 Problem . . . 20
3.4.2 Answer . . . 20
3.5 Exercise 1.39: Lambert's continued fraction for tangents . . . 21
3.5.1 Problem . . . 21
3.5.2 Answer . . . 21

**4 Section 1.3.4** **23**
4.1 Exercise 1.40: Approximating cubics with Newton's method . 23
4.1.1 Problem . . . 23
4.1.2 Answer . . . 23
4.2 Exercise 1.41: Doubling `double` . . . 25
4.2.1 Problem . . . 25
4.2.2 Answer . . . 25
4.3 Exercise 1.42: Composing functions . . . 26
4.3.1 Problem . . . 26
4.3.2 Answer . . . 26
4.4 Exercise 1.43: Repeated function application . . . 27
4.4.1 Problem . . . 27
4.4.2 Answer . . . 27
4.5 Exercise 1.44: Repeated smoothing . . . 28
4.5.1 Problem . . . 28
4.5.2 Answer . . . 28
4.6 **TODO** Exercise 1.45: $n$th roots with average damping . . . 30
4.6.1 Problem . . . 30
4.6.2 Answer . . . 31
4.7 Exercise 1.46: Generalizing iterative improvement . . . 31
4.7.1 Problem . . . 31
4.7.2 Answer . . . 31

# 1 Section 1.3.1

## 1.1 Exercise 1.29: Implementing Simpson's Rule

### 1.1.1 Problem

Simpson's Rule is a more accurate method of numerical integration than the method illustrated above. Using Simpson's Rule, the integral of a function $f$ between $a$ and $b$ is approximated as

$$\frac{h}{3}(y_0 + 4y_1 + 2y_2 + 4y + 3 + 2y_4 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n)$$

where $h = (b - a)/n$, for some even integer $n$, and $y_k = f(a + kh)$. (Increasing $n$ increases the accuracy of the approximation.) Define a procedure that takes as arguments $f$, $a$, $b$, and $n$ and returns the value of the integral, computed using Simpson's Rule. Use your procedure to integrate `cube` between 0 and 1 (with $n = 100$ and $n = 1000$), and compare the results to those of the `integral` procedure shown above.

### 1.1.2 Answer

Between the text and the exercise statement, it's not too hard to come up with a procedure to implement Simpson's Rule. First, the relevant code from the text:

```
(require (planet neil/sicp:1:17))

(define dx 0.00001)

(define (square x) (* x x))
(define (cube x) (* x x x))

(define (inc n) (+ n 1))

(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

```
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))
```

The only bit here that takes a bit of thought is coming up with a specification for the $k$th term of the rule (the trickiest bit is captured in `term-mult`):

```
(define (simpsons-integral f a b n)
  (define h (/ (- b a) n))
  (define (y k) (f (+ a (* k h))))
  (define (term-mult k)
    (cond ((or (= k 0) (= k n)) 1)
          ((odd? k) 4)
          ((even? k) 2)))
  (define (simpson-term k)
    (* (term-mult k) (y k)))
  (cond ((or (odd? n) (<= n 0))
         (printf "n must be even and greater than zero)~n"))
        ((<= b a)
         (printf "a must be greater than b~n"))
        (else
         (* (/ h 3) (sum simpson-term 0.0 inc n)))))
```

To compare the results, we'll stack up each function against the other.

```
(define (time-once f)
  (define t0 (current-milliseconds))
  (define r1 (f))
  (define t1 (current-milliseconds))
  (printf "~a (~a ms)~n" r1 (- t1 t0)))

(define (exercise f start increment end)
  (define (iter i)
    (define t0 (current-milliseconds))
    (define r1 (f i))
    (define t1 (current-milliseconds))

    (printf "~a: ~a (~a ms)~n"
            i
```

```
          r1 (- t1 t0))
    (if (< i end)
        (iter (+ i increment)))) ; Simpson's rule needs even n

  (iter start))
```

For a simple evaluation, we can look at the value of the text's `integrate` when applied to the `square` function. As a definite integral,

$$\int_1^4 x^2 \mathrm{d}x = \left[\frac{x^3}{3} + c\right]_1^4 = \left(\frac{4^3}{3} + c)\right) - \left(\frac{1^3}{3} + c)\right) = \frac{64}{3} - \frac{1}{3} = 21$$

The procedure from the text does reasonably well, though it takes around 100 ms to execute on my machine. Slow!

```
(time-once (lambda () (integral square 1 4 dx)))

21.000000000151953 (100 ms)
```

Our procedure for Simpson's Rule does quite well, with some numerical error depending on the intervals selected. It is extremely fast, though:

```
(define simpsons-n2
  (lambda (n) (simpsons-integral square 1 4 n)))
(exercise simpsons-n2 2 8 40)

2: 21.0 (0 ms)
10: 21.0 (0 ms)
18: 20.999999999999996 (0 ms)
26: 21.000000000000007 (0 ms)
34: 20.999999999999996 (0 ms)
42: 21.0 (0 ms)
```

Now let's try $x^3$. For an exact solution,

$$\int_{-2}^3 x^3 \mathrm{d}x = \left[\frac{x^4}{4}\right]_{-2}^3 = \left(\frac{(3)^4}{4}\right) - \left(\frac{(-2)^4}{4}\right) = \frac{81}{4} - \frac{16}{4} = \frac{65}{4} = 16.25$$

For the `integral` procedure from the text, we get a result that's very close, but it takes the better part of a second to execute:

```
(time-once (lambda () (integral cube -2 3 dx)))
```

```
16.250000000258588 (155 ms)
```

Comparing this to our `simpsons-integral` method over the same range, we get an even closer result, with similar numeric fluctuation depending on the subdivisions:

```
(define simpsons-n3
  (lambda (n) (simpsons-integral cube -2 3 n)))
(exercise simpsons-n3 2 4 20)
```

```
2: 16.25 (0 ms)
6: 16.250000000000004 (0 ms)
10: 16.25 (0 ms)
14: 16.250000000000007 (0 ms)
18: 16.25 (0 ms)
22: 16.250000000000004 (0 ms)
```

Trying something a bit trickier, we can look at the cosine function. The text's `integral` procedure comes up with a very good approximation, but takes over a full *second* to do so on my machine!

```
(time-once (lambda () (integral cos 0 10 dx)))
```

```
-0.54402111069616 (699 ms)
```

Simpson's Rule, again, comes up with its approximatation in a fraction of the time.

```
(define simpsons-cos
  (lambda (n) (simpsons-integral cos 0 10 n)))
(exercise simpsons-cos 2 50 500)
```

```
2: 2.1592953546274214 (0 ms)
52: -0.5440252627727152 (0 ms)
102: -0.5440213904266393 (0 ms)
152: -0.5440211675384317 (1 ms)
202: -0.5440211290472229 (0 ms)
252: -0.5440211183852472 (0 ms)
302: -0.5440211145232634 (0 ms)
352: -0.5440211128582283 (0 ms)
402: -0.5440211120467381 (1 ms)
452: -0.5440211116134976 (0 ms)
502: -0.5440211113653068 (0 ms)
```

## 1.2 Exercise 1.30: Iterative `sum`

### 1.2.1 Problem

The `sum` procedure above generates a linear recursion. The procedure can be rewritten so that the sum is performed iteratively. Show how to do this by filling in the missing expressions in the following definition:

```
(define (sum term a next b)
  (define (iter a result)
    (if <??>
        <??>
        (iter <??> <??>)))
  (iter <??> <??>))
```

### 1.2.2 Answer

This exercise echoes the many similar conversions between procedures that generate iterative and recursive processes in the previous section. As a fill-in-the-blank problem, this is pretty straightforward.

```
(define (sum-iter term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ result (term a)))))
  (iter a 0))
```

Evaluating it, we can see that it works just as it should:

```
(sum-iter identity 1 inc 10)
```

```
55
```

For the example with `cube` from the text, which should result in 3025:

```
(sum-iter cube 1 inc 10)
```

```
3025
```

And the approximation to $\pi$:

```
(* 8 (sum-iter (lambda (x) (/ 1.0 (* x (+ x 2))) )
               1
               (lambda (x) (+ x 4))
               1000))
```

```
3.139592655589782
```

## 1.3 Exercise 1.31: A `product` procedure

### 1.3.1 Problem

a. The `sum` procedure is only the simplest of a vast number of similar abstractions that can be captured as higher-order procedures. Write an analogous procedure called `product` that returns the product of the values of a function at points over a given range. Show how to define `factorial` in terms of `product`. Also use `product` to compute approximations to $\pi$ using the formula

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 9 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

b. If your `product` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

### 1.3.2 Answer

This exercise has us starting to develop a pattern: a number of specific instances that will lead to the development of an abstraction to capture them all. The basic structure of the `product` procedures is straightforward: substitute the base case (1 instead of 0) and the operation (`*` instead of `+`, of course). The recursive case is straightforward.

```
(define (product-recur term a next b)
  (if (> a b)
      1
      (* (term a)
         (product-recur term (next a) next b))))
```

The iterative case is just as easy to adapt from `sum-iter`.

```
(define (product-iter term a next b)
  (define (iter a result)
```

```
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1))
```

Using this for factorial is dirt simple: we just need to use the `identity` function and `inc`:

```
(define (factorial-recur n) (product-recur identity 1 inc n))
(define (factorial-iter n) (product-iter identity 1 inc n))
(printf "recursive: ~a~n" (factorial-recur 10))
(printf "iterative: ~a~n" (factorial-iter 10))
```

```
recursive: 3628800
iterative: 3628800
```

We have to be a bit more thoughtful in coming up with the function that generates each term of the aproximation. The easiest way to do this is to break it down into very small functions that can be composed to specify the final numerators and denominators in a straightforward way:

```
(define (approx-pi n product-func)
  (define (round-up-to-even i)
    (if (even? i) i (inc i)))
  (define (round-up-to-odd i)
    (if (odd? i) i (inc i)))
  (define (num k)
    (round-up-to-even (inc k)))
  (define (den k)
    (round-up-to-odd (inc k)))
  (define (term k)
    (/ (num k) (den k)))
  ;; Use a real number here to switch Racket's exact rationals
  ;; to inexact floating point
  (* 4.0 (product-func term 1 inc n)))

(printf "recursive: ~a~n" (approx-pi 1000 product-recur))
(printf "iterative: ~a~n" (approx-pi 1000 product-iter))
```

```
recursive: 3.1431607055322663
iterative: 3.1431607055322663
```

## 1.4 Exercise 1.32: Abstracting `accumulate`

### 1.4.1 Problem

a. Show that `sum` and `product` (Exercise 1.31) are both special cases of a still more general notion called `accumulate` that combines a collection of terms, using some general accumulation function:

```
(accumulate combiner null-value term a next b)
```

`accumulate` takes as arguments the same term and range specifications as `sum` and `product`, together with a `combiner` procedure (of two arguments) that specifies how the current term is to be combined with the accumulation of the preceding terms and a `null-value` that specifies what base value to use when the terms run out. Write `accumulate` and show how `sum` and `product` can both be defined as simple calls to `accumulate`.

b. If your `accumulate` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

### 1.4.2 Answer

Now we're cooking with gas! This exercise is a key one on the path toward thinking in terms of appropriate abstractions. The exercise description and the process of solving the last two exercises should be enough to put you on the track to solving this one: instead of specific procedure names, the exact `combiner` and `null-value` are supplied in the procedure call.

Creating procedures for the recursive and iterative forms requires the same substitutions:

```
(define (accumulate-recur combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
                (accumulate-recur combiner null-value term
                                  (next a) next b))))

(define (accumulate-iter combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (term a)))))
```

```
  (iter a null-value))
```

Testing these out, we can see that both variants work identically to the versions from the previous exercises.

```
(define (sum-acc-recur term a next b)
  (accumulate-recur + 0 term a next b))

(define (sum-acc-iter term a next b)
  (accumulate-iter + 0 term a next b))

(define (product-acc-recur term a next b)
  (accumulate-recur * 1 term a next b))

(define (product-acc-iter term a next b)
  (accumulate-iter * 1 term a next b))

(printf "sum-acc-recur: ~a~n"
        (sum-acc-iter identity 1 inc 10))
(printf "sum-acc-iter:  ~a~n"
        (sum-acc-recur identity 1 inc 10))
(printf "product-acc-recur: ~a~n"
        (product-acc-recur identity 1 inc 10))
(printf "product-acc-iter:  ~a~n"
        (product-acc-iter identity 1 inc 10))
```

```
sum-acc-recur: 55
sum-acc-iter:  55
product-acc-recur: 3628800
product-acc-iter:  3628800
```

## 1.5 Exercise 1.33: Further generalization of accumulation

### 1.5.1 Problem

You can obtain an even more general version of `accumulate` (Exercise 1.32) by introducing the notion of a "filter" on the terms to be combined. That is, combine only those terms derived from values in the range that satisfy a specified condition. The resulting `filtered-accumulate` abstraction takes the same arguments as accumulate, together with an additional predicate of one argument that specifies the filter. Write `filtered-accumulate` as a procedure. Show how to express the following using `filtered-accumulate`:

a. the sum of the squares of the prime numbers in the interval $a$ to $b$ (assuming that you have a `prime?` predicate already written)

b. the product of all the positive integers less than $n$ that are relatively prime to $n$ (i.e., all positive integers $i < n$ such that GCD$(i, n) = 1$).

### 1.5.2 Answer

These are the early steps in the "build your own lightsaber" aspect of the book. Variants of many of the functions used in these exercises can be found in every Lisp. The form of the `filtered-accumulate` function may not be exactly the usual form (there are a lot of arguments that need to be supplied, imposing a more specific shape on the functions used), but the underlying principle is one that will be encountered again and again.

The key point to notice here is in how the higher-order functions are used as tools of generalization. Just adding in a `filter` predicate allows easier expression of a whole range of procedures.

```
(define (filtered-accumulate combiner null-value filter
                             term a next b)
  (define (iter a result)
    (cond ((> a b) result)
          ((filter a) (iter (next a) (combiner result (term a))))
          (else (iter (next a) result))))
  (iter a null-value))
```

Transplanting the prime-detection code from the last section, it's very straightforward to build the target `sum-of-squares-of-primes` procedure:

```
(define (square n) (* n n))

(define (smallest-divisor n)
  (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (divides? a b)
  (= (remainder b a) 0))
```

```
;; Fixed this procedure to no longer classify 1 as prime
(define (prime? n)
  (and (= n (smallest-divisor n))
       (> n 1)))

(define (sum-of-squares-of-primes a b)
  (filtered-accumulate + 0 prime? square a inc b))

(printf "sum of squares of primes between 2 and 6: ~a~n"
  (sum-of-squares-of-primes 2 6))

sum of squares of primes between 2 and 6: 38
```

With the `gcd` procedure in place, the `product-of-relative-primes` procedure is similarly succinct:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))

(define (product-of-relative-primes n)
  (define (relatively-prime? i)
    (= (gcd i n) 1))
  (filtered-accumulate * 1 relatively-prime? identity 1 inc n))

(printf "product of relative primes of 6: ~a~n"
  (product-of-relative-primes 6))
(printf "product of relative primes of 7: ~a~n"
  (product-of-relative-primes 7))

product of relative primes of 6: 5
product of relative primes of 7: 720
```

# 2 Section 1.3.2

## 2.1 Exercise 1.34: Perverse self-application

### 2.1.1 Problem

Suppose we define the procedure

```
(define (f g)
   (g 2))
```

Then we have

```
(f square)
4

(f (lambda (z) (* z (+ z 1))))
6
```

What happens if we (perversely) ask the interpreter to evaluate the combination `(f f)`? Explain.

#### 2.1.2 Answer

To solve this one, simply go through the substitution steps that were outline in section 1.1. This procedure call expands as:

```
(f f)
(f 2)
(2 2)
```

And it will terminate with an error, since `2` is not a procedure.

# 3 Section 1.3.3

## 3.1 TODO Exercise 1.35: The fixed point $\phi$

### 3.1.1 Problem

Show that the golden ratio $\phi$ (section 1.22) is a fixed point of the transformation $x \mapsto 1 + 1/x$, and use this fact to compute $\phi$ by means of the `fixed-point` procedure.

### 3.1.2 Answer

## 3.2 Exercise 1.36: Observing `fixed-point` approximations

### 3.2.1 Problem

Modify `fixed-point` so that it prints the sequence of approximations it generates, using the `newline` and `display` primitives shown in Exercise

1.22. Then find a solution to $x^x = 1000$ by finding a fixed point of $x \mapsto \log 1000 / \log x$. (Use Scheme's primitive `log` procedure, which computes natural logarithms.) Compare the number of steps this takes with and without average damping. (Note that you cannot start `fixed-point` with a guess of 1, as this would cause division by $\log 1 = 0$.)

### 3.2.2 Answer

First, the definitions from the text:

```
(define (average x y)
  (/ (+ x y) 2))

(define tolerance 0.00001)

(define (fixed-point-orig f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

Now, we instrument `fixed-point` so we can observe its progress:

```
(define (fixed-point-verbose f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (display next)
      (newline)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

First, we examine the sequence generated by the basic fixed-point search, starting from 1.1 (since, as the exercise tells us, we can't use 1 as a starting point).

```
(fixed-point-verbose
 (lambda (x) (/ (log 1000) (log x))) 1.1)

72.47657378429035
1.6127318474109593
14.45350138636525
2.5862669415385087
7.269672273367045
3.4822383620848467
5.536500810236703
4.036406406288111
4.95053682041456
4.318707390180805
4.721778787145103
4.450341068884912
4.626821434106115
4.509360945293209
4.586349500915509
4.535372639594589
4.568901484845316
4.546751100777536
4.561341971741742
4.551712230641226
4.558059671677587
4.55387226495538
4.556633177654167
4.554812144696459
4.556012967736543
4.555220997683307
4.555743265552239
4.555398830243649
4.555625974816275
4.555476175432173
4.555574964557791
4.555509814636753
4.555552779647764
4.555524444961165
4.555543131130589
4.555530807938518
4.555538934848503
```

When average damping is introduce, the convergence is greatly accelerated:

```
(fixed-point-verbose (lambda (x) (average x (/ (log 1000) (log x)))) 1.1)
```

Running a quick comparison of these sequences, we can see that average damping, in this case, reduces the number of steps required by almost two-thirds.

```
(prin1 "Without damping.:")
(print (- (length (split-string normal "\n")) 1))
(prin1 "With average damping:")
(print (- (length (split-string damped "\n")) 1))
```

```
"Without damping.:"
37
"With average damping:"
13
```

## 3.3 Exercise 1.37: Infinite continued fractions

### 3.3.1 Problem

a. An infinite "continued fraction" is an expression of the form

$$f = \cfrac{N_1}{D_1 + \cfrac{N_2}{D_2 + \cfrac{N_3}{D_3 + \cdots}}}$$

As an example, one can show that the infinite continued fraction expansion with the $N_i$ and the $D_i$ all equal to 1 produces $1/\phi$, where $\phi$ is the golden ratio (described in section 1.2.2). One way to approximate an infinite continued fraction is to truncate the expansion after a given number of terms. Such a truncation—a so-called finite continued fraction "$k$-term finite continued fraction"—has the form

$$\cfrac{N_1}{D_1 + \cfrac{N_2}{\cdots + \cfrac{N_K}{D_K + \cdots}}}$$

Suppose that `n` and `d` are procedures of one argument (the term index $i$) that return the $N_i$ and $D_i$ of the terms of the continued fraction. Define a procedure `cont-frac` such that evaluating `(cont-frac n d k)` computes the value of the $k$-term finite continued fraction. Check your procedure by approximating $1/\phi$ using

```
(cont-frac (lambda (i) 1.0)
           (lambda (i) 1.0)
            k)
```

for successive values of `k`. How large must you make `k` in order to get an approximation that is accurate to 4 decimal places?

b. If your `cont-frac` procedure generates a recursive process, write one that generates an iterative process. If it generates an iterative process, write one that generates a recursive process.

### 3.3.2 Answer

Writing this procedure in recursive form is pleasingly straightforward, as it's very close to the mathematical definition. There is one tricky bit, though: the normal recursive specification counts *down* from $k$ to 1, rather than *up* from 1 to $k$.

```
(define (cont-frac-recur n d k)
  (define (recur i)
    (if (< i k)
        (/ (n i) (+ (d i) (recur (inc i))))
        (/ (n 1) (d 1))))
  (recur 1))

(define cont-frac cont-frac-recur)
```

As seen in previous exercises, $1/\phi \approx 0.61803398875$. The specification of `approx-pi` is dirt simple, as long as we remember that we have to supply lambdas: just supplying '1.0' isn't sufficient, as `cont-frac` will try to call whatever is supplied for `n` and `d` as procedures. These lambdas ignore their single input and just return a constant value.

```
(define (approx-phi k)
  (cont-frac-recur (lambda (i) 1.0) (lambda (i) 1.0) k))
(exercise approx-phi 1 1 15)

1: 1.0 (0 ms)
2: 0.5 (0 ms)
3: 0.6666666666666666 (0 ms)
4: 0.6000000000000001 (0 ms)
5: 0.625 (0 ms)
```

```
6: 0.6153846153846154 (0 ms)
7: 0.6190476190476191 (0 ms)
8: 0.6176470588235294 (0 ms)
9: 0.6181818181818182 (0 ms)
10: 0.6179775280898876 (0 ms)
11: 0.6180555555555556 (0 ms)
12: 0.6180257510729613 (0 ms)
13: 0.6180371352785146 (0 ms)
14: 0.6180327868852459 (0 ms)
15: 0.6180344478216819 (0 ms)
```

We can see that it takes expansion to 12 terms for the fraction to reach an accuracy of four decimal places.

Writing the continued fraction procedure in iterative style is not quite as natural from the mathematical perspective, but it shouldn't be too hard if you've gone spent the time doing the many similar conversions in previous exercises.

```
(define (cont-frac-iter n d k)
  (define (iter i result)
    (if (= i 0)
        result
        (iter (- i 1) (/ (n i) (+ (d i) result)))))
  (iter k 0))

(define (approx-phi-iter k)
  (cont-frac-iter (lambda (i) 1.0) (lambda (i) 1.0) k))
```

The iterative version goes through exactly the same sequence of approximations:

```
(exercise approx-phi-iter 1 1 15)

1: 1.0 (0 ms)
2: 0.5 (0 ms)
3: 0.6666666666666666 (0 ms)
4: 0.6000000000000001 (0 ms)
5: 0.625 (0 ms)
6: 0.6153846153846154 (0 ms)
7: 0.6190476190476191 (0 ms)
8: 0.6176470588235294 (0 ms)
```

```
9: 0.6181818181818182 (0 ms)
10: 0.6179775280898876 (0 ms)
11: 0.6180555555555556 (0 ms)
12: 0.6180257510729613 (0 ms)
13: 0.6180371352785146 (0 ms)
14: 0.6180327868852459 (0 ms)
15: 0.6180344478216819 (0 ms)
```

## 3.4 Exercise 1.38: Euler's continued fraction for $e - 2$

### 3.4.1 Problem

In 1737, the Swiss mathematician Leonhard Euler published a memoir *De Fractionibus Continuis*, which included a continued fraction expansion for $e-2$, where $e$ is the base of the natural logarithms. In this fraction, the $N_i$ are all 1, and the $D_i$ are successively $1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots$ Write a program that uses your `cont-frac` procedure from Exercise 1-37 to approximate $e$, based on Euler's expansion.

### 3.4.2 Answer

The last exercise required most of the heavy lifting, when we created the `cont-frac` procedure. There's a bit of thought required to identify the pattern for the denominators of this sequence, but it's not too bad: basically, for each block of three terms, the middle term is twice the number of the block. This can be specified with judicious use of `remainder`, `/`, and `*`.

```
(define (euler-e k)
  (+ 2.0 (cont-frac (lambda (i) 1)
                    (lambda (i)
                      (if (= (remainder i 3) 2)
                          (* 2 (/ (+ i 1) 3))
                          1))
                    k)))
```

We recall that $e \approx 2.7182818$. Running the approximation through an increasing number of terms, we see that it takes about nine terms to achieve an accuracy of four decimal places.

```
(exercise euler-e 1 1 20)
```

```
1: 3.0 (0 ms)
2: 2.5 (0 ms)
3: 2.75 (0 ms)
4: 2.7142857142857144 (0 ms)
5: 2.7272727272727275 (0 ms)
6: 2.717948717948718 (0 ms)
7: 2.7183098591549295 (0 ms)
8: 2.7181818181818183 (0 ms)
9: 2.718283582089552 (0 ms)
10: 2.7182817182817183 (0 ms)
11: 2.718282368249837 (0 ms)
12: 2.7182818229439496 (0 ms)
13: 2.718281828735696 (0 ms)
14: 2.7182818267351814 (0 ms)
15: 2.718281828470584 (0 ms)
16: 2.7182818284585633 (0 ms)
17: 2.7182818284626897 (0 ms)
18: 2.7182818284590278 (0 ms)
19: 2.718281828459046 (0 ms)
20: 2.7182818284590398 (0 ms)
```

## 3.5 Exercise 1.39: Lambert's continued fraction for tangents

### 3.5.1 Problem

A continued fraction representation of the tangent function was published in 1770 by the German mathematician J.H. Lambert:

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cdots}}}$$

where $x$ is in radians. Define a procedure `(tan-cf x k)` that computes an approximation to the tangent function based on Lambert's formula. `k` specifies the number of terms to compute, as in Exercise 1.37.

### 3.5.2 Answer

Having gone through the previous exercises, this one poses few challenges. The only tricky bits here are making sure that all of the $N_i$ are negative for $i > 1$ and finding a tidy expression for the $D_i$.

```
(define (tan-cf x k)
```

```
(cont-frac (lambda (i)
             (if (= i 1)
                 x
                 (- (* x x))))
           (lambda (i)
             (+ 1.0 (* 2.0 (- i 1.0))))
           k))
```

With this achieved, we can see that the approximation works just as it should. Checking against $\tan 1 \approx 1.5574077$:

```
(exercise (lambda (k) (tan-cf 1 k)) 1 1 15)

1: 1.0 (0 ms)
2: 0.5 (0 ms)
3: 1.3333333333333333 (0 ms)
4: 1.5454545454545454 (0 ms)
5: 1.5571428571428574 (0 ms)
6: 1.5574043261231283 (0 ms)
7: 1.5574076959027885 (0 ms)
8: 1.5574077244820586 (0 ms)
9: 1.5574077246541251 (0 ms)
10: 1.5574077246548994 (0 ms)
11: 1.557407724654902 (0 ms)
12: 1.557407724654902 (0 ms)
13: 1.557407724654902 (0 ms)
14: 1.557407724654902 (0 ms)
15: 1.557407724654902 (0 ms)
```

And against $\tan 3 \approx -0.14254654$:

```
(exercise (lambda (k) (tan-cf 3 k)) 1 1 15)

1: 3.0 (0 ms)
2: 0.75 (0 ms)
3: -6.0 (0 ms)
4: -0.7894736842105263 (0 ms)
5: -0.2946428571428571 (0 ms)
6: -0.16600790513833985 (0 ms)
7: -0.14471968709256838 (0 ms)
8: -0.14267962042476284 (0 ms)
```

```
9: -0.14255240897388746 (0 ms)
10: -0.1425467394618813 (0 ms)
11: -0.1425465482543813 (0 ms)
12: -0.14254654318489302 (0 ms)
13: -0.14254654307623113 (0 ms)
14: -0.14254654307430684 (0 ms)
15: -0.14254654307427825 (0 ms)
```

Easy peasy.

# 4 Section 1.3.4

## 4.1 Exercise 1.40: Approximating cubics with Newton's method

### 4.1.1 Problem

Define a procedure `cubic` that can be used together with the `newtons-method` procedure in expressions of the form

```
(newtons-method (cubic a b c) 1)
```

to approximate zeros of the cubic $x^3 + ax^2 + bx + c$.

### 4.1.2 Answer

First, a look at the code from the text:

```
(define fixed-point fixed-point-verbose)

(define (average-damp f)
  (lambda (x) (average x (f x))))

(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))

(define dx 0.00001)

(define (deriv g)
  (lambda (x)
```

```
    (/ (- (g (+ x dx)) (g x))
       dx)))

(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))

(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
```

Creating the `cubic` procedure is straightforward: we just need to return a new lambda with bindings to the appropriate coefficients:

```
(define (cubic a b c)
  (lambda (x) (+ (cube x) (* a (square x)) (* b x) c)))
```

To test this out, we can select a cubic with easy-to-identify roots:

$$(x-1)(x-5)(x-9) = x^3 - 15x^2 + 59x - 45$$

The zeros, clearly, are when $x = 1$, $x = 5$, and $x = 9$. Putting this under the lens:

```
(define cubic-1-5-9 (cubic -15 59 -45))
(newtons-method cubic-1-5-9 0)

0.7627138035042597
0.9814506189483658
0.9998724196890224
0.9999999943750526
1.000000000000021
```

With a starting guess of 0, it homes in on the zero at x=1. With a different starting point:

```
(define cubic-1-5-9 (cubic -15 59 -45))
(newtons-method cubic-1-5-9 20)

15.242795357171445
12.19425210065894
10.347170447494172
9.38219704852933
```

```
9.044810267042461
9.000733970524449
9.000000204681681
9.000000000000787
```

It slides down to 9. An appropriate starting guess will also let it find the root at $x = 5$:

```
(define cubic-1-5-9 (cubic -15 59 -45))
(newtons-method cubic-1-5-9 6)
```

```
4.846151183366139
5.000457174131245
4.999999999987518
4.999999999999999
```

But, of course, this procedure doesn't guarantee that it will find *all* the zeros: it will just find one; which one is found depends on the choice of the initial guess.

## 4.2 Exercise 1.41: Doubling `double`

### 4.2.1 Problem

Define a procedure `double` that takes a procedure of one argument as argument and returns a procedure that applies the original procedure twice. For example, if `inc` is a procedure that adds 1 to its argument, then `(double inc)` should be a procedure that adds 2. What value is returned by

```
(((double (double double)) inc) 5)
```

### 4.2.2 Answer

This exercise again attempts to get you thinking in terms of procedures that return other procedures. It shouldn't take too much effort to come up with the desire specification.

```
(define (double f)
  (lambda (x) (f (f x))))
```

`double` isn't a very good name for this procedure; `twice` would be a better description of its actual semantics. Consider what would happen if we were to call:

```
(define (divide-by-two x) (/ x 2.0))
((double divide-by-two) 16.0)
```

Confusing! Changing the name would better capture the actual semantics:

```
(define twice double)
((twice divide-by-two) 16.0)
```

The reader of the code would have an actual fighting chase of figuring out what was actually going on.

Now, looking at the expression that the exercises asks us to evaluate, it shouldn't be too hard to trace out what's going on, as long as you look carefully at what is returning a procedure and what is done with that procedure once it is returned.

```
(((double (double double)) inc) 5)
```

The result is an explosive 16 ($2^{2^2}$) calls to `inc`.

## 4.3 Exercise 1.42: Composing functions

### 4.3.1 Problem

Let $f$ and $g$ be two one-argument functions. The *composition* $f$ after $g$ is defined to be the function $x \mapsto f(g(x))$. Define a procedure `compose` that implements composition. For example, if `inc` is a procedure that adds 1 to its argument,

```
((compose square inc) 6)
49
```

### 4.3.2 Answer

Here's another place where the authors have you create a standard tool of functional programming: function composition. The specification of the procedure couldn't be simpler:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

The example from the text works exactly as expected:

```
((compose square inc) 6)

49
```

The exercise doesn't do much to point out why this might be useful, as the example is extremely basic. In general, it can be used to build up chains of composite operations without mucking around with the boilerplate needed to make them call each other. This will become more important as the books moves beyond simple numeric manipulations into operations on complex data structures.

## 4.4 Exercise 1.43: Repeated function application

### 4.4.1 Problem

If $f$ is a numerical function and $n$ is a positive integer, then we can form the $n$th repeated application of $f$, which is defined to be the function whose value at $x$ is $f(f(\ldots(f(x))\ldots))$. For example, if $f$ is the function $x \mapsto x+1$, then the $n$th repeated application of $f$ is the function $x \mapsto x + n$. If $f$ is the operation of squaring a number, then the nth repeated application of f is the function that raises its argument to the $2^n$th power. Write a procedure that takes as inputs a procedure that computes $f$ and a positive integer $n$ and returns the procedure that computes the $n$th repeated application of $f$. Your procedure should be able to be used as follows:

```
((repeated square 2) 5)
625
```

Hint: You may find it convenient to use `compose` from Exercise 1-42.

### 4.4.2 Answer

This exercise again attempts to draw out the flexibility provided by higher-order functions. A recursive specification of `repeated` is easy to produce, particularly with the hint given that `compose` may ease the creation

```
(define (repeated f n)
  (if (> n 1)
      (compose f (repeated f (dec n)))
      f))
```

As it should, the example from the exercise results in $(5^2)^2 = 25^2 = 625$:

```
((repeated square 2) 5)
```

```
625
```

It's easy to apply this to other procedures:

```
((repeated (lambda (n) (/ n 2)) 4) 240)
```

```
15
```

This works with higher-order procedures, as well (though using nested invocations where the result is called immediately can result in code that is excessively difficult to untangle. Compare this to the results from exercise 1.41, as it can be tricky to tease out the reason for what at first looks like it might result in the same behavior:

```
(((repeated twice 3) inc) 5)
```

```
13
```

## 4.5 Exercise 1.44: Repeated smoothing

### 4.5.1 Problem

The idea of "smoothing" a function is an important concept in signal processing. If $f$ is a function and $dx$ is some small number, then the smoothed version of $f$ is the function whose value at a point $x$ is the average of $f(x-dx)$, $f(x)$, and $f(x+dx)$. Write a procedure `smooth` that takes as input a procedure that computes $f$ and returns a procedure that computes the smoothed $f$. It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtained the "$n$-fold smoothed function". Show how to generate the $n$-fold smoothed function of any given function using `smooth` and `repeated` from Exercise 1.43.

### 4.5.2 Answer

Once again, all that's needed for the design of this procedure is a careful reading of the exercise.

```
(define (smooth f)
  (lambda (x)
    (average (f x) (f (- x dx)))))

(define (repeated-smooth f n)
  ((repeated smooth n) f))
```

To test out this function, we need a messy function. For this, an extremely short-period trigonometic function will do nicely. (We could use deterministic random noise, but that's much more of a pain to implement.)

```
(define (wobble x) (* 10 (sin (* 4000000 x))))
(define (noisy-square x) (+ (square x) (wobble x)))

(exercise square 1.0 1.0 10)

1.0: 1.0 (0 ms)
2.0: 4.0 (0 ms)
3.0: 9.0 (0 ms)
4.0: 16.0 (0 ms)
5.0: 25.0 (0 ms)
6.0: 36.0 (0 ms)
7.0: 49.0 (0 ms)
8.0: 64.0 (0 ms)
9.0: 81.0 (0 ms)
10.0: 100.0 (0 ms)
```

Adding in the noise makes the function fluctuate wildly.

```
(exercise noisy-square 1.0 1.0 10)

1.0: -8.901405464041472 (0 ms)
2.0: 1.2260722922389635 (0 ms)
3.0: 18.124275898362253 (0 ms)
4.0: 21.330138724433453 (0 ms)
5.0: 17.368988825278407 (0 ms)
6.0: 28.531995756103797 (0 ms)
7.0: 54.538812853073914 (0 ms)
8.0: 73.01973005666515 (0 ms)
9.0: 77.98810912901972 (0 ms)
10.0: 90.13647382064232 (0 ms)
```

A single smoothing doesn't have much of an effect.

```
(exercise (smooth noisy-square) 1.0 1.0 10)

1.0: -1.170768490150289 (0 ms)
2.0: 7.1173970760167204 (0 ms)
```

```
3.0: 12.04408830114544 (0 ms)
4.0: 13.735367720702044 (0 ms)
5.0: 21.321394972443517 (0 ms)
6.0: 37.23396689626534 (0 ms)
7.0: 53.0242037654625 (0 ms)
8.0: 63.89331330110201 (0 ms)
9.0: 76.94576984070798 (0 ms)
10.0: 98.97071928376823 (0 ms)
```

Multiple smoothings, on the other hand (in this case, 6), begin to recreate the shape of the original function: the resulting values are within 0.1 of the original `square` procedure.

```
(exercise (repeated-smooth noisy-square 6) 1.0 1.0 10)
```

```
1.0: 0.9589530488442973 (0 ms)
2.0: 4.0152118485939905 (0 ms)
3.0: 9.045102245275618 (0 ms)
4.0: 15.997114201795398 (0 ms)
5.0: 24.953676520451445 (0 ms)
6.0: 35.989392103222414 (0 ms)
7.0: 49.0427324823916 (0 ms)
8.0: 64.02185727993435 (0 ms)
9.0: 80.96256542416378 (0 ms)
10.0: 99.96672650555777 (0 ms)
```

## 4.6 TODO Exercise 1.45: $n$th roots with average damping

### 4.6.1 Problem

We saw in section 1.3.3 that attempting to compute square roots by naively finding a fixed point of $y \mapsto x/y$ does not converge, and that this can be fixed by average damping. The same method works for finding cube roots as fixed points of the average-damped $y \mapsto x/y^2$. Unfortunately, the process does not work for fourth roots—a single average damp is not enough to make a fixed-point search for $y \mapsto x/y^3$ converge. On the other hand, if we average damp twice (*i.e.*, use the average damp of the average damp of $y \mapsto x/y^3$) the fixed-point search does converge. Do some experiments to determine how many average damps are required to compute $n$th roots as a fixed-point search based upon repeated average damping of $y \mapsto x/y^(n-1)$. Use this to implement a simple procedure for computing $n$th roots using `fixed-point`,

`average-damp`, and the `repeated` procedure of Exercise 1-43. Assume that any arithmetic operations you need are available as primitives.

### 4.6.2 Answer

## 4.7 Exercise 1.46: Generalizing iterative improvement

### 4.7.1 Problem

Several of the numerical methods described in this chapter are instances of an extremely general computational strategy known as "iterative improvement". Iterative improvement says that, to compute something, we start with an initial guess for the answer, test if the guess is good enough, and otherwise improve the guess and continue the process using the improved guess as the new guess. Write a procedure `iterative-improve` that takes two procedures as arguments: a method for telling whether a guess is good enough and a method for improving a guess. `iterative-improve` should return as its value a procedure that takes a guess as argument and keeps improving the guess until it is good enough. Rewrite the `sqrt` procedure of section 1.1.7 and the `fixed-point` procedure of section 1.3.3 in terms of `iterative-improve`.

### 4.7.2 Answer

This one can be a bit tricky to get right, and the details of the exercise specification are important: particularly, note that `iterative-improve` is supposed to return a procedure. Otherwise, the structure is simple: we just need to generalize the structure of the procedures that we've already created. This is one exercise I definitely recommend fighting your way through, yourself!

```
(define (iterative-improve good-enough? improve)
  (lambda (initial-guess)
    (define (iter guess last-guess)
      (if (good-enough? guess last-guess)
          guess
          (iter (improve guess) guess)))
    (iter (improve initial-guess) initial-guess)))
```

The version of `sqrt` below is adapted from the improved version from exercise 1.7, as both this version and the `fixed-point` procedure use subsequent guesses as arguments to the `good-enough?` predicate.

Note that the when `sqrt-ii`'s `improve` is created, it captures (closes over) the value of `x`, and this is used by the procedure when it is later executed by `iterative-improve`.

```
(define (sqrt-ii x)
  (define tolerance 0.001)
  (define (percent-changed new old)
    (/ (- new old) old))
  (define (small-change? new-guess old-guess)
    (< (abs (percent-changed new-guess old-guess)) tolerance))
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (average a b)
    (/ (+ a b) 2))
  (define (improve guess)
    (average guess (/ x guess)))
  ((iterative-improve small-change? improve) 1.0))
```

Testing, we can see that this specialization of the general iterative improvement abstraction works just as it should:

```
(printf "sqrt-ii(4) = ~a~n" (sqrt-ii 4))
(printf "sqrt-ii(9) = ~a~n" (sqrt-ii 9))
(printf "sqrt-ii(16) = ~a~n" (sqrt-ii 16))
(printf "sqrt-ii(100) = ~a~n" (sqrt-ii 100))
```

```
sqrt-ii(4) = 2.0000000929222947
sqrt-ii(9) = 3.000000001396984
sqrt-ii(16) = 4.000000636692939
sqrt-ii(100) = 10.000000000139897
```

```
(define (fixed-point-ii f first-guess)
  (define tolerance 0.00001)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (improve guess)
    (f guess))
  ((iterative-improve close-enough? improve) first-guess))
```

We can test this out on the average-damped approach to finding a solution to $x^x = 1000$, as developed in exercise 1.36. The value calculated as a result of that exercise was 4.555536364911781.

```
(fixed-point-ii
 (lambda (x) (average x (/ (log 1000) (log x)))) 1.1)

4.555536364911781
```

Happily, this version based off of the generalized `iterative-improve` produces the same result.