*4.5 NUMERICAL ANALYSIS

Accumulator algorithms are also used in the natural and social sciences to approximate the values of common mathematical constants, and to numerically compute values and roots of complicated functions that cannot be solved mathematically. In this section, we will discuss a few relatively simple examples from this field of mathematics and computer science, known as *numerical analysis*.

The harmonic series

Suppose we have an ant that starts walking from one end of a 1 meter long rubber rope. During the first minute, the ant walks 10 cm. At the end of the first minute, we stretch the rubber rope uniformly by 1 meter, so it is now 2 meters long. During the next minute, the ant walks another 10 cm, and then we stretch the rope again by 1 meter. If we continue this process indefinitely, will the ant ever reach the other end of the rope? If it does, how long will it take?

The answer lies in counting what fraction of the rope the ant traverses in each minute. During the first minute, the ant walks 1/10 of the distance to the end of the rope. After stretching, the ant has *still* traversed 1/10 of the distance because the portion of the rope on which the ant walked was doubled along with the rest of the rope. However, in the second minute, the ant's 10 cm stroll only covers $10/200 = 1/20$ of the entire distance. Therefore, after 2 minutes, the ant has covered $1/10 + 1/20$ of the rope. During the third minute, the rope is 3 m long, so the ant covers only $10/300 = 1/30$ of the distance. This pattern continues, so our problem boils down to whether the following sum ever reaches 1.

$$\frac{1}{10} + \frac{1}{20} + \frac{1}{30} + \cdots = \frac{1}{10}\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots\right)$$

Naturally, we can answer this question using an accumulator. But how do we add these fractional terms? In Exercise 4.1.23, you may have computed $1 + 2 + 3 + \cdots + n$:

```
def sumNumbers(n):
    total = 0
    for number in range(1, n + 1):
        total = total + number
    return total
```

In each iteration of this `for` loop, we add the value of `number` to the accumulator variable `total`. Since `number` is assigned the values $1, 2, 3, \ldots,$ `n`, `total` has the sum of these values after the loop. To compute the fraction of the rope traveled by the ant, we can modify this function to add `1 / number` in each iteration instead, and then multiply the result by `1/10`:

```
def ant(n):
    """Simulates the "ant on a rubber rope" problem.  The rope
    is initially 1 m long and the ant walks 10 cm each minute.

    Parameter:
        n: the number of minutes the ant walks

    Return value: fraction of the rope traveled by the ant in n minutes
    """

    total = 0
    for number in range(1, n + 1):
        total = total + (1 / number)
    return total * 0.1
```

To answer our question with this function, we need to try several values of n to see if we can find a sufficiently high value for which the sum exceeds 1. If we find such a value, then we need to work with smaller values until we find the value of n for which the sum *first* reaches or exceeds 1.

Reflection 1 *Using at most 5 calls to the* `ant` *function, find a range of minutes that answers the question.*

For example, `ant(100)` returns about `0.52`, `ant(1000)` returns about `0.75`, `ant(10000)` returns about `0.98`, and `ant(15000)` returns about `1.02`. So the ant will reach the other end of the rope after 10,000–15,000 minutes. As cumbersome as that was, continuing it to find the exact number of minutes required would be far worse.

Reflection 2 *How would we write a function to find when the ant first reaches or exceeds the end of the rope? (Hint: this is similar to the carbon-14 half-life problem.) We will leave the answer as an exercise.*

This infinite version of the sum that we computed in our loop,

$$1 + \frac{1}{2} + \frac{1}{3} + \cdots,$$

is called the *harmonic series* and each finite sum

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

is called the n^{th} harmonic number. (Mathematically, the ant reaches the other end of the rope because the harmonic series *diverges*, that is, its partial sums increase forever.) The harmonic series can be used to approximate the natural logarithm (ln) function. For sufficiently large values of n,

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n + 0.577.$$

This is illustrated in Figure 1 for only small values of n. Notice how the approximation improves as n increases.

Reflection 3 *Knowing that* $H_n \approx \ln n + 0.577$*, how can you approximate how long until the ant will reach the end of the rope if it walks only 1 cm each minute?*

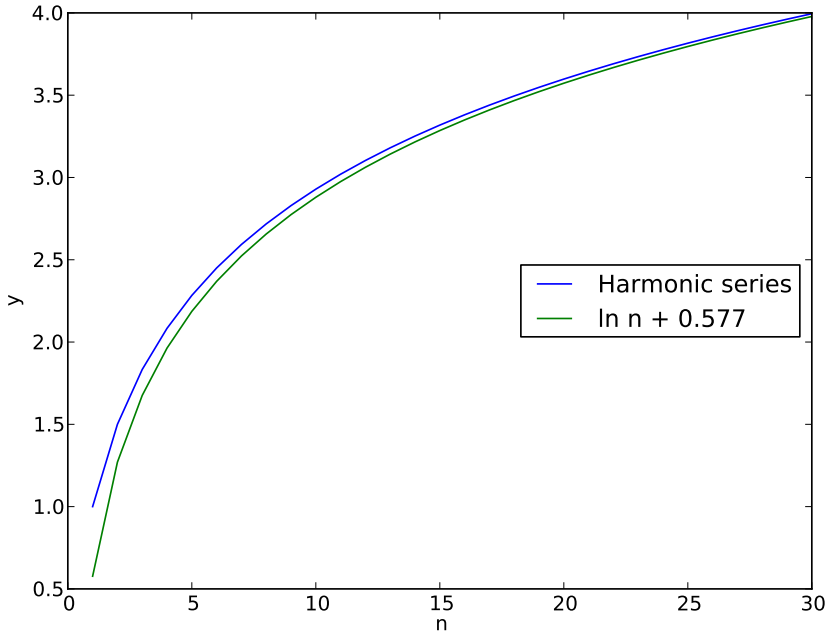

Figure 1 Harmonic series approximation of the natural logarithm (ln).

To answer this question, we want to find n such that

$$100 = \ln n + 0.577$$

since the ant's first step is not 1/100 of the total distance. This is the same as

$$e^{100-0.577} = n.$$

In Python, we can find the answer with `math.exp(100 - 0.577)`, which gives about 1.5×10^{43} minutes, a long time indeed.

Approximating π

The value π is probably the most famous mathematical constant. There have been many infinite series found over the past 500 years that can be used to approximate π. One of the most famous is known as the *Leibniz series*, named after Gottfried Leibniz, the co-inventor of calculus:

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$$

Like the harmonic series approximation of the natural logarithm, the more terms we compute of this series, the closer we get to the true value of π. To compute this sum, we need to identify a pattern in the terms, and relate them to the values of the index variable in a `for` loop. Then we can fill in the red blank line below with an expression that computes the i^{th} term from the value of the index variable `i`.

```
def leibniz(terms):
    """Computes a partial sum of the Leibniz series.

    Parameter:
        terms: the number of terms to add

    Return value: the sum of the given number of terms
    """

    total = 0
    for i in range(terms):
        total = total + ____________
    pi = total * 4
    return pi
```

To find the pattern, we can write down the values of the index variable next to the values in the series to identify a relationship:

i	0	1	2	3	4	⋯
i^{th} term	1	$-\frac{1}{3}$	$\frac{1}{5}$	$-\frac{1}{7}$	$\frac{1}{9}$	⋯

Ignoring the alternating signs for a moment, we can see that the absolute value of the i^{th} term is

$$\frac{1}{2i+1}.$$

To alternate the signs, we use the fact that -1 raised to an even power is 1, while -1 raised to an odd power is -1. Since the even terms are positive and odd terms are negative, the final expression for the i term is

$$(-1)^i \cdot \frac{1}{2i+1}.$$

Therefore, the red assignment statement in our `leibniz` function should be

```
total = total + (-1) ** i / (2 * i + 1)
```

Reflection 4 *Call the completed* `leibniz` *function with a series of increasing arguments. What do you notice about how the values converge to* π*?*

By examining several values of the function, you might notice that they alternate between being greater and less than the actual value of π. Figure 2 illustrates this.

Approximating square roots

The square root function ($\sqrt{n}$) cannot, in general, be computed directly. But it can be approximated very well with many iterations of the following difference equation, known as the *Babylonian method* (or *Newton's method*):

$$X(k) = \frac{1}{2}\left(X(k-1) + \frac{n}{X(k-1)}\right),$$

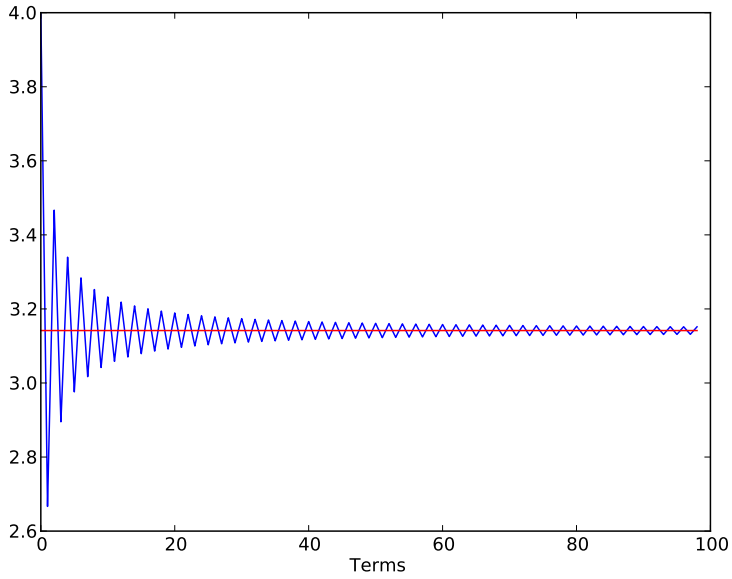

Figure 2 The Leibniz series converging to π.

where n is the value whose square root we want and k is the number of iterations of the algorithm. If $k = 0$, then $X(k)$ is defined to be 1. The approximation of $\sqrt{n}$ will be better for larger values of k; $X(20)$ will be closer to the actual square root than $X(10)$.

Similar to our previous examples, we can compute successive values of the difference equation using iteration. In this case, each value is computed from the previous value according to the formula above. If we let the variable name `x` represent a term in the difference equation, then we can compute the $\mathtt{k}^{\mathrm{th}}$ term with the following simple function:

```
def sqrt(n, k):
    """Approximates the square root of n with k iterations of the
       Babylonian method.

    Parameters:
        n: the number to take the square root of
        k: number of iterations

    Return value: the approximate square root of n
    """

    x = 1.0
    for index in range(k):
        x = 0.5 * (x + n / x)
    return x
```

Reflection 5 *Call the function above to approximate* $\sqrt{10}$ *with various values of* k*. What value of* k *is necessary to match the value given by the* `math.sqrt` *function?*

Exercises

4.5.1* Recall Reflection 2: How would we write a function to find when the ant first reaches or exceeds the end of the rope? (Hint: this is similar to the carbon-14 half-life problem.)

(a) Write a function to answer this question.

(b) How long does it take for the ant to traverse the entire rope?

(c) If the ant walks 5 cm each minute, how long does it take to reach the other end?

4.5.2. Augment the `ant` function so that it also produces the plot in Figure 1.

4.5.3* The value e (Euler's number, the base of the natural logarithm) is equal to the infinite sum

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

Write a function

```
e(n)
```

that approximates the value of e by computing and returning the value of `n` terms of this sum. For example, calling `e(4)` should return the value $1 + 1/1 + 1/2 + 1/6 \approx 2.667$. Your function should *call* the `factorial` function you wrote for Exercise 4.1.26 to aid in the computation.

4.5.4. Calling the `factorial` function repeatedly in the function you wrote for the previous problem is very inefficient because many of the same arithmetic operations are being performed repeatedly. Explain where this is happening.

4.5.5. To avoid the problems suggested by the previous exercise, rewrite the function from Exercise 4.5.3 without calling the `factorial` function.

4.5.6* Rather than specifying the number of iterations in advance, numerical algorithms usually iterate until the absolute value of the current term is sufficiently small. At this point, we assume the approximation is "good enough." Rewrite the `leibniz` function so that it iterates while the absolute value of the current term is greater than 10^{-6}.

4.5.7. Similar to the previous exercise, rewrite the `sqrt` function so that it iterates while the absolute value of the difference between the current and previous values of `x` is greater than 10^{-15}.

4.5.8. The following expression, discovered in the 14th century by Indian mathematician Madhava of Sangamagrama, is another way to compute π.

$$\pi = \sqrt{12}\left(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \cdots\right)$$

Write a function

```
approxPi(n)
```

that computes `n` terms of this expression to approximate π. For example, `approxPi(3)` should return the value

$$\sqrt{12}\left(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2}\right) \approx \sqrt{12}(1 - 0.111 + 0.022) \approx 3.156.$$

To determine the pattern in the sequence of terms, consider this table:

Term number	Term
0	$1/(3^0 \cdot 1)$
1	$1/(3^1 \cdot 3)$
2	$1/(3^2 \cdot 5)$
3	$1/(3^3 \cdot 7)$
⋮	⋮
i	?

What is the term for a general value of i?

4.5.9* The Wallis product, named after 17th century English mathematician John Wallis, is an infinite product that converges to π:

$$\pi = 2\left(\frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \cdots\right)$$

Write a function

```
wallis(terms)
```

that computes the given number of terms in the Wallis product. Hint: Consider the terms in pairs and find an expression for each pair. Then iterate over the number of pairs needed to flesh out the required number of terms. You may assume that `terms` is even.

4.5.10. The Nilakantha series, named after Nilakantha Somayaji, a 15th century Indian mathematician, is another infinite series for π:

$$\pi = 3 + \frac{4}{2 \cdot 3 \cdot 4} - \frac{4}{4 \cdot 5 \cdot 6} + \frac{4}{6 \cdot 7 \cdot 8} - \frac{4}{8 \cdot 9 \cdot 10} + \cdots$$

Write a function

```
nilakantha(terms)
```

that computes the given number of terms in the Nilakantha series.

4.5.11. The following infinite product was discovered by François Viète, a 16th century French mathematician:

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdot \frac{2}{\sqrt{2+\sqrt{2}}} \cdot \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \cdot \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2+\sqrt{2}}}}} \cdots$$

Write a function

```
viete(terms)
```

that computes the given number of terms in the Viète's product. (Look at the pattern carefully; it is not as hard as it looks if you base the denominator in each term on the denominator of the previous term.)

Selected Exercise Solutions

4.5.1 (a)
```
def ant():
    total = 0
    step = 0
    while total < 10:
        step = step + 1
        total = total + (1 / step)
    return step
```

(b) 12,367 minutes = over 8.5 days

(c) about 272,459,350 minutes = over 518 years

4.5.3
```
def e(n):
    total = 1
    for i in range(1, n):
        total = total + 1.0 / factorial(i)
    return total
```

4.5.6
```
def leibniz():
    total = 0
    term = 1
    index = 0
    while abs(term) > 1e-6:
        term = (-1) ** index / (2 * index + 1)
        total = total + term
        index = index + 1
    pi = total * 4
    return pi
```

4.5.9
```
def wallis(terms):
    pairs = terms / 2
    product = 1
    for i in range(pairs):
        first = (2.0 * (i + 1)) / (2 * i + 1)
        second = (2.0 * (i + 1)) / (2 * i + 3)
        product = product * first * second
    pi = product * 2
    return pi
```