Chapter One

Numerical Algorithms

> The word "algorithm" derives from the name of the Persian mathematician (Abu Ja'far Muhammad ibn Musa) Al-Khwarizmi who lived from about 790 CE to about 840 CE. He wrote a book, *Hisab al-jabr w'al-muqabala*, that also named the subject "algebra."

Numerical analysis is the subject which studies algorithms for computing expressions defined with real numbers. The square-root $\sqrt{y}$ is an example of such an expression; we evaluate this today on a calculator or in a computer program as if it were as simple as y^2. It is numerical analysis that has made this possible, and we will study how this is done. But in doing so, we will see that the same approach applies broadly to include functions that cannot be named, and it even changes the nature of fundamental questions in mathematics, such as the impossibility of finding expressions for roots of order higher than 4.

There are two different phases to address in numerical analysis:

- the development of algorithms and
- the analysis of algorithms.

These are in principle independent activities, but in reality the development of an algorithm is often guided by the analysis of the algorithm, or of a simpler algorithm that computes the same thing or something similar.

There are three characteristics of algorithms using real numbers that are in conflict to some extent:

- the *accuracy* (or *consistency*) of the algorithm,
- the *stability* of the algorithm, and
- the effects of finite-precision arithmetic (a.k.a. round-off error).

The first of these just means that the algorithm approximates the desired quantity to any required accuracy under suitable restrictions. The second means that the behavior of the algorithm is continuous with respect to the parameters of the algorithm. The third topic is still not well understood at the most basic level, in the sense that there is not a well-established mathematical model for finite-precision arithmetic. Instead, we are forced to use crude upper bounds for the behavior of finite-precision arithmetic

that often lead to overly pessimistic predictions about its effects in actual computations.

We will see that in trying to improve the accuracy or efficiency of a stable algorithm, one is often led to consider algorithms that turn out to be unstable and therefore of minimal (if any) value. These various aspects of numerical analysis are often intertwined, as ultimately we want an algorithm that we can analyze rigorously to ensure it is effective when using computer arithmetic.

The *efficiency* of an algorithm is a more complicated concept but is often the bottom line in choosing one algorithm over another. It can be related to all of the above characteristics, as well as to the *complexity* of the algorithm in terms of computational work or memory references required in its implementation.

Another central theme in numerical analysis is *adaptivity*. This means that the computational algorithm adapts itself to the data of the problem being solved as a way to improve efficiency and/or stability. Some adaptive algorithms are quite remarkable in their ability to elicit information automatically about a problem that is required for more efficient solution.

We begin with a problem from antiquity to illustrate each of these components of numerical analysis in an elementary context. We will not always disentangle the different issues, but we hope that the differing components will be evident.

1.1 FINDING ROOTS

People have been computing roots for millennia. Evidence exists [64] that the Babylonians, who used base-60 arithmetic, were able to approximate

$$\sqrt{2} \approx 1 + \frac{24}{60} + \frac{51}{60^2} + \frac{10}{60^3} \tag{1.1}$$

nearly 4000 years ago. By the time of Heron[1] a method to compute square-roots was established [26] that we recognize now as the Newton-Raphson-Simpson method (see section 2.2.1) and takes the form of a repeated iteration

$$x \leftarrow \tfrac{1}{2}(x + y/x), \tag{1.2}$$

where the backwards arrow $\leftarrow$ means *assignment* in algorithms. That is, once the computation of the expression on the right-hand side of the arrow has been completed, a new value is assigned to the variable x. Once that assignment is completed, the computation on the right-hand side can be redone with the new x.

The algorithm (1.2) is an example of what is known as *fixed-point iteration*, in which one hopes to find a fixed point, that is, an x where the iteration quits changing. A *fixed point* is thus a point x where

$$x = \tfrac{1}{2}(x + y/x). \tag{1.3}$$

[1] A.k.a. Hero, of Alexandria, who lived in the 1st century CE.

More precisely, x is a fixed point $x = f(x)$ of the function

$$f(x) = \tfrac{1}{2}(x + y/x), \tag{1.4}$$

defined, say, for $x \neq 0$. If we rearrange terms in (1.3), we find $x = y/x$, or $x^2 = y$. Thus a fixed point as defined in (1.3) is a solution of $x^2 = y$, so that $x = \pm\sqrt{y}$.

To describe actual implementations of these algorithms, we choose the scripting syntax implemented in the system *octave*. As a programming language, this has some limitations, but its use is extremely widespread. In addition to the public domain implementation of `octave`, a commercial interpreter (which predates `octave`) called Matlab is available. However, all computations presented here were done in `octave`.

We can implement (1.2) in `octave` in two steps as follows. First, we define the function (1.4) via the code

```
function x=heron(x,y)
x=.5*(x+y/x);
```

To use this function, you need to start with some initial guess, say, $x = 1$, which is written simply as

```
x=1
```

(Writing an expression with and without a semicolon at the end controls whether the interpreter prints the result or not.) But then you simply iterate:

```
x=heron(x,y)
```

until x (or the part you care about) quits changing. The results of doing so are given in table 1.1.

We can examine the accuracy by a simple code

```
function x=errheron(x,y)
for i=1:5
    x=heron(x,y);
    errheron=x-sqrt(y)
end
```

We show in table 1.1 the results of these computations in the case $y = 2$. This algorithm seems to "home in" on the solution. We will see that the accuracy doubles at each step.

1.1.1 Relative versus absolute error

We can require the accuracy of an algorithm to be based on the size of the answer. For example, we might want the approximation $\hat{x}$ of a root x to be small relative to the size of x:

$$\frac{\hat{x}}{x} = 1 + \delta, \tag{1.5}$$

$\sqrt{2}$ approximation	absolute error
1.**5**0000000000000	8.5786e-02
1.41**6**666666666667	2.4531e-03
1.414215**6**8627451	2.1239e-06
1.41421356237**4**69	1.5947e-12
1.41421356237309	-2.2204e-16

Table 1.1 Results of experiments with the Heron algorithm applied to approximate $\sqrt{2}$ using the algorithm (1.2) starting with $x = 1$. The boldface indicates the leading incorrect digit. Note that the number of correct digits essentially doubles at each step.

where δ satisfies some fixed tolerance, e.g., $|\delta| \leq \epsilon$. Such a requirement is in keeping with the model we will adopt for floating-point operations (see (1.31) and section 18.1).

We can examine the relative accuracy by the simple code

```
function x=relerrher(x,y)
for i=1:6
    x=heron(x,y);
    errheron=(x/sqrt(y))-1
end
```

We leave as exercise 1.2 comparison of the results produced by the above code `relerrher` with the absolute errors presented in table 1.1.

1.1.2 Scaling Heron's algorithm

Before we analyze how Heron's algorithm (1.2) works, let us enhance it by a prescaling. To begin with, we can suppose that the number y whose square root we seek lies in the interval $[\frac{1}{2}, 2]$. If $y < \frac{1}{2}$ or $y > 2$, then we make the transformation

$$\tilde{y} = 4^k y \tag{1.6}$$

to get $\tilde{y} \in [\frac{1}{2}, 2]$, for some integer k. And of course $\sqrt{\tilde{y}} = 2^k \sqrt{y}$. By scaling y in this way, we limit the range of inputs that the algorithm must deal with.

In table 1.1, we showed the absolute error for approximating $\sqrt{2}$, and in exercise 1.2 the relative errors for approximating $\sqrt{2}$ and $\sqrt{\frac{1}{2}}$ are explored. It turns out that the maximum errors for the interval $[\frac{1}{2}, 2]$ occur at the ends of the interval (exercise 1.3). Thus five iterations of Heron, preceded by the scaling (1.6), are sufficient to compute $\sqrt{y}$ to 16 decimal places.

Scaling provides a simple example of adaptivity for algorithms for finding roots. Without scaling, the global performance (section 1.2.2) would be quite different.

1.2 ANALYZING HERON'S ALGORITHM

As the name implies, a major objective of numerical analysis is to analyze the behavior of algorithms such as Heron's iteration (1.2). There are two questions one can ask in this regard. First, we may be interested in the local behavior of the algorithm assuming that we have a reasonable start near the desired root. We will see that this can be done quite completely, both in the case of Heron's iteration and in general for algorithms of this type (in chapter 2). Second, we may wonder about the global behavior of the algorithm, that is, how it will respond with arbitrary starting points. With the Heron algorithm we can give a fairly complete answer, but in general it is more complicated. Our point of view is that the global behavior is really a different subject, e.g., a study in dynamical systems. We will see that techniques like scaling (section 1.1.2) provide a basis to turn the local analysis into a convergence theory.

1.2.1 Local error analysis

Since Heron's iteration (1.2) is recursive in nature, it it natural to expect that the errors can be expressed recursively as well. We can write an algebraic expression for Heron's iteration (1.2) linking the error at one iteration to the error at the next. Thus define

$$x_{n+1} = \tfrac{1}{2}(x_n + y/x_n), \tag{1.7}$$

and let $e_n = x_n - x = x_n - \sqrt{y}$. Then by (1.7) and (1.3),

$$\begin{aligned} e_{n+1} &= x_{n+1} - x = \tfrac{1}{2}(x_n + y/x_n) - \tfrac{1}{2}(x + y/x) \\ &= \tfrac{1}{2}(e_n + y/x_n - y/x) = \tfrac{1}{2}\left(e_n + \frac{y(x - x_n)}{x x_n}\right) \\ &= \tfrac{1}{2}\left(e_n - \frac{x e_n}{x_n}\right) = \tfrac{1}{2} e_n \left(1 - \frac{x}{x_n}\right) = \tfrac{1}{2}\frac{e_n^2}{x_n}. \end{aligned} \tag{1.8}$$

If we are interested in the relative error,

$$\hat{e}_n = \frac{e_n}{x} = \frac{x_n - x}{x} = \frac{x_n}{x} - 1, \tag{1.9}$$

then (1.8) becomes

$$\hat{e}_{n+1} = \tfrac{1}{2}\frac{x\hat{e}_n^2}{x_n} = \tfrac{1}{2}\left(1 + \hat{e}_n\right)^{-1} \hat{e}_n^2. \tag{1.10}$$

Thus we see that

the error at each step is proportional to
the square of the error at the previous step;

for the relative error, the constant of proportionality tends rapidly to $\frac{1}{2}$. In (2.20), we will see that this same result can be derived by a general technique.

1.2.2 Global error analysis

In addition, (1.10) implies a limited type of *global convergence* property, at least for $x_n > x = \sqrt{y}$. In that case, (1.10) gives

$$|\hat{e}_{n+1}| = \tfrac{1}{2}\frac{\hat{e}_n^2}{|1+\hat{e}_n|} = \tfrac{1}{2}\frac{\hat{e}_n^2}{1+\hat{e}_n} \leq \tfrac{1}{2}\hat{e}_n. \tag{1.11}$$

Thus the relative error is reduced by a factor smaller than $\frac{1}{2}$ at each iteration, no matter how large the initial error may be. Unfortunately, this type of global convergence property does not hold for many algorithms. We can illustrate what can go wrong in the case of the Heron algorithm when $x_n < x = \sqrt{y}$.

Suppose for simplicity that $y = 1$, so that also $x = 1$, so that the relative error is $\hat{e}_n = x_n - 1$, and therefore (1.10) implies that

$$\hat{e}_{n+1} = \tfrac{1}{2}\frac{(1-x_n)^2}{x_n}. \tag{1.12}$$

As $x_n \to 0$, $\hat{e}_{n+1} \to \infty$, even though $|\hat{e}_n| < 1$. Therefore, convergence is not truly global for the Heron algorithm.

What happens if we start with x_0 near zero? We obtain x_1 near ∞. From then on, the iterations satisfy $x_n > \sqrt{y}$, so the iteration is ultimately convergent. But the number of iterations required to reduce the error below a fixed error tolerance can be arbitrarily large depending on how small x_0 is. By the same token, we cannot bound the number of required iterations for arbitrarily large x_0. Fortunately, we will see that it is possible to choose good starting values for Heron's method to avoid this potential bad behavior.

1.3 WHERE TO START

With any iterative algorithm, we have to start the iteration somewhere, and this choice can be an interesting problem in its own right. Just like the initial scaling described in section 1.1.2, this can affect the performance of the overall algorithm substantially.

For the Heron algorithm, there are various possibilities. The simplest is just to take $x_0 = 1$, in which case

$$\hat{e}_0 = \frac{1}{x} - 1 = \frac{1}{\sqrt{y}} - 1. \tag{1.13}$$

This gives

$$\hat{e}_1 = \tfrac{1}{2}x\hat{e}_0^2 = \tfrac{1}{2}x\left(\frac{1}{x} - 1\right)^2 = \tfrac{1}{2}\frac{(x-1)^2}{x}. \tag{1.14}$$

We can use (1.14) as a formula for $\hat{e}_1$ as a function of x (it is by definition a function of $y = x^2$); then we see that

$$\hat{e}_1(x) = \hat{e}_1(1/x) \tag{1.15}$$

by comparing the rightmost two terms in (1.14). Note that the maximum of $\hat{e}_1(x)$ on $[2^{-1/2}, 2^{1/2}]$ occurs at the ends of the interval, and

$$\hat{e}_1(\sqrt{2}) = \tfrac{1}{2}\frac{(\sqrt{2}-1)^2}{\sqrt{2}} = \tfrac{3}{4}\sqrt{2} - 1 \approx 0.060660\,. \tag{1.16}$$

Thus the simple starting value $x_0 = 1$ is remarkably effective. Nevertheless, let us see if we can do better.

1.3.1 Another start

Another idea to start the iteration is to make an approximation to the square-root function given the fact that we always have $y \in [\frac{1}{2}, 2]$ (section 1.1.2). Since this means that y is near 1, we can write $y = 1 + t$ (i.e., $t = y - 1$), and we have

$$\begin{aligned} x = \sqrt{y} = \sqrt{1+t} &= 1 + \tfrac{1}{2}t + \mathcal{O}(t^2) \\ &= 1 + \tfrac{1}{2}(y-1) + \mathcal{O}(t^2) = \tfrac{1}{2}(y+1) + \mathcal{O}(t^2). \end{aligned} \tag{1.17}$$

Thus we get the approximation $x \approx \frac{1}{2}(y+1)$ as a possible starting guess:

$$x_0 = \tfrac{1}{2}(y+1). \tag{1.18}$$

But this is the same as x_1 if we had started with $x_0 = 1$. Thus we have not really found anything new.

1.3.2 The best start

Our first attempt (1.18) based on a linear approximation to the square-root did not produce a new concept since it gives the same result as starting with a constant guess after one iteration. The approximation (1.18) corresponds to the tangent line of the graph of $\sqrt{y}$ at $y = 1$, but this may not be the best affine approximation to a function on an interval. So let us ask the question, What is the best approximation to $\sqrt{y}$ on the interval $[\frac{1}{2}, 2]$ by a linear polynomial? This problem is a miniature of the questions we will address in chapter 12.

The general linear polynomial is of the form

$$f(y) = a + by. \tag{1.19}$$

If we take $x_0 = f(y)$, then the relative error $\hat{e}_0 = \hat{e}_0(y)$ is

$$\hat{e}_0(y) = \frac{x_0 - \sqrt{y}}{\sqrt{y}} = \frac{a + by - \sqrt{y}}{\sqrt{y}} = \frac{a}{\sqrt{y}} + b\sqrt{y} - 1. \tag{1.20}$$

Let us write $e_{ab}(y) = \hat{e}_0(y)$ to be precise. We seek a and b such that the maximum of $|e_{ab}(y)|$ over $y \in [\frac{1}{2}, 2]$ is minimized.

Fortunately, the functions

$$e_{ab}(y) = \frac{a}{\sqrt{y}} + b\sqrt{y} - 1 \tag{1.21}$$

have a simple structure. As always, it is helpful to compute the derivative:

$$e'_{ab}(y) = -\tfrac{1}{2}ay^{-3/2} + \tfrac{1}{2}by^{-1/2} = \tfrac{1}{2}(-a + by)y^{-3/2}. \tag{1.22}$$

Thus $e'_{ab}(y) = 0$ for $y = a/b$; further, $e'_{ab}(y) > 0$ for $y > a/b$, and $e'_{ab}(y) < 0$ for $y < a/b$. Therefore, e_{ab} has a minimum at $y = a/b$ and is strictly increasing as we move away from that point in either direction. Thus we have proved that

$$\min e_{ab} = \min e_{ba} = e_{ab}(a/b) = 2\sqrt{ab} - 1. \tag{1.23}$$

Thus the maximum values of $|e_{ab}|$ on $[\frac{1}{2}, 2]$ will be at the ends of the interval or at $y = a/b$ if $a/b \in [\frac{1}{2}, 2]$. Moreover, the best value of $e_{ab}(a/b)$ will be negative (exercise 1.10). Thus we consider the three values

$$\begin{aligned} e_{ab}(2) &= \frac{a}{\sqrt{2}} + b\sqrt{2} - 1 \\ e_{ab}(\tfrac{1}{2}) &= a\sqrt{2} + \frac{b}{\sqrt{2}} - 1 \\ -e_{ab}(a/b) &= 1 - 2\sqrt{ab}. \end{aligned} \tag{1.24}$$

Note that $e_{ab}(2) = e_{ba}(1/2)$. Therefore, the optimal values of a and b must be the same: $a = b$ (exercise 1.11). Moreover, the minimum value of e_{ab} must be minus the maximum value on the interval (exercise 1.12). Thus the optimal value of $a = b$ is characterized by

$$a\tfrac{3}{2}\sqrt{2} - 1 = 1 - 2a \implies a = \left(\tfrac{3}{4}\sqrt{2} + 1\right)^{-1}. \tag{1.25}$$

Recall that the simple idea of starting the Heron algorithm with $x_0 = 1$ yielded an error

$$|\hat{e}_1| \le \gamma = \tfrac{3}{4}\sqrt{2} - 1, \tag{1.26}$$

and that this was equivalent to choosing $a = \frac{1}{2}$ in the current scheme. Note that the optimal $a = 1/(\gamma + 2)$, only slightly less than $\frac{1}{2}$, and the resulting minimum value of the maximum of $|e_{aa}|$ is

$$1 - 2a = 1 - \frac{2}{\gamma + 2} = \frac{\gamma}{\gamma + 2}. \tag{1.27}$$

Thus the optimal value of a reduces the previous error of γ (for $a = \frac{1}{2}$) by nearly a factor of $\frac{1}{2}$, despite the fact that the change in a is quite small. The benefit of using the better initial guess is of course squared at each iteration, so the reduced error is nearly smaller by a factor of 2^{-2^k} after k iterations of Heron. We leave as exercise 1.13 the investigation of the effect of using this optimal starting place in the Heron algorithm.

1.4 AN UNSTABLE ALGORITHM

Heron's algorithm has one drawback in that it requires division. One can imagine that a simpler algorithm might be possible such as

$$x \leftarrow x + x^2 - y. \tag{1.28}$$

n	0	1	2	3	4	5
x_n	1.5	1.75	2.81	8.72	82.8	6937.9
n	6	7	8	9	10	11
x_n	5×10^{7}	2×10^{15}	5×10^{30}	3×10^{61}	8×10^{122}	7×10^{245}

Table 1.2 Unstable behavior of the iteration (1.28) for computing $\sqrt{2}$.

Before experimenting with this algorithm, we note that a fixed point

$$x = x + x^2 - y \tag{1.29}$$

does have the property that $x^2 = y$, as desired. Thus we can assert the *accuracy* of the algorithm (1.28), in the sense that any fixed point will solve the desired problem. However, it is easy to see that the algorithm is not *stable*, in the sense that if we start with an initial guess with any sort of error, the algorithm fails. table 1.2 shows the results of applying (1.28) starting with $x_0 = 1.5$. What we see is a rapid movement *away* from the solution, followed by a catastrophic blowup (which eventually causes failure in a fixed-precision arithmetic system, or causes the computer to run out of memory in a variable-precision system). The error is again being squared, as with the Heron algorithm, but since the error is getting bigger rather than smaller, the algorithm is useless. In section 2.1 we will see how to diagnose instability (or rather how to guarantee stability) for iterations like (1.28).

1.5 GENERAL ROOTS: EFFECTS OF FLOATING-POINT

So far, we have seen no adverse effects related to finite-precision arithmetic. This is common for (stable) iterative methods like the Heron algorithm. But now we consider a more complex problem in which rounding plays a dominant role.

Suppose we want to compute the roots of a general quadratic equation $x^2 + 2bx + c = 0$, where $b < 0$, and we chose the algorithm

$$x \leftarrow -b + \sqrt{b^2 - c}. \tag{1.30}$$

Note that we have assumed that we can compute the square-root function as part of this algorithm, say, by Heron's method.

Unfortunately, the simple algorithm in (1.30) fails if we have $c = \epsilon^2 b^2$ (it returns $x = 0$) as soon as $\epsilon^2 = c/b^2$ is small enough that the floating-point representation of $1 - \epsilon^2$ is 1. For any (fixed) finite representation of real numbers, this will occur for some $\epsilon > 0$.

We will consider floating-point arithmetic in more detail in section 18.1, but the simple model we adopt says that the result of computing a binary operator $\oplus$ such as $+$, $-$, $/$, or $*$ has the property that

$$fl(a \oplus b) = (a \oplus b)(1 + \delta), \tag{1.31}$$

where $|\delta| \leq \epsilon$, where $\epsilon > 0$ is a parameter of the model.[2] However, this means that a collection of operations could lead to catastrophic cancellation, e.g., $f\ell(f\ell(1 + \frac{1}{2}\epsilon) - 1) = 0$ and not $\frac{1}{2}\epsilon$.

We can see the behavior in some simple codes. But first, let us simplify the problem further so that we have just one parameter to deal with. Suppose that the equation to be solved is of the form

$$x^2 - 2bx + 1 = 0. \tag{1.32}$$

That is, we switch b to $-b$ and set $c = 1$. In this case, the two roots are multiplicative inverses of each other. Define

$$x_\pm = b \pm \sqrt{b^2 - 1}. \tag{1.33}$$

Then $x_- = 1/x_+$.

There are various possible algorithms. We could use one of the two formulas $x_\pm = b \pm \sqrt{b^2 - 1}$ directly. More precisely, let us write $\tilde{x}_\pm \approx b \pm \sqrt{b^2 - 1}$ to indicate that we implement this in floating-point. Correspondingly, there is another pair of algorithms that start by computing $\tilde{x}_\mp$ and then define, say, $\hat{x}_+ \approx 1/\tilde{x}_-$. A similar algorithm could determine $\hat{x}_- \approx 1/\tilde{x}_+$.

All four of these algorithms will have different behaviors. We expect that the behaviors of the algorithms for computing $\tilde{x}_-$ and $\hat{x}_-$ will be dual in some way to those for computing $\tilde{x}_+$ and $\hat{x}_+$, so we consider only the first pair.

First, the function `minus` implements the $\tilde{x}_-$ square-root algorithm:

```
function x=minus(b)
% solving = 1-2bx +x^2
x=b-sqrt(b^2-1);
```

To know if it is getting the right answer, we need another function to `check` the answer:

```
function error=check(b,x)
error = 1-2*b*x +x^2;
```

To automate the process, we put the two together:

```
function error=chekminus(b)
x=minus(b);
error=check(b,x)
```

For example, when $b = 10^6$, we find the error is -7.6×10^{-6}. As b increases further, the error increases, ultimately leading to complete nonsense. For this reason, we consider an alternative algorithm suitable for large b.

The algorithm for $\hat{x}_-$ is given by

[2] The notation $f\ell$ is somewhat informal. It would be more precise to write $a \,\widehat{\oplus}\, b$ instead of $f\ell(a \oplus b)$ since the operator is modified by the effect of rounding.

```
function x=plusinv(b)
% solving = 1-2bx +x^2
y=b+sqrt(b^2-1);
x=1/y;
```

Similarly, we can check the accuracy of this computation by the code

```
function error=chekplusinv(b)
x=plusinv(b);
error=check(b,x)
```

Now when $b = 10^6$, we find the error is -2.2×10^{-17}. And the bigger b becomes, the more accurate it becomes.

Here we have seen that algorithms can have data-dependent behavior with regard to the effects of finite-precision arithmetic. We will see that there are many algorithms in numerical analysis with this property, but suitable analysis will establish conditions on the data that guarantee success.

1.6 EXERCISES

Exercise 1.1 *How accurate is the approximation (1.1) if it is expressed as a decimal approximation (how many digits are correct)?*

Exercise 1.2 *Run the code* `relerrher` *starting with* $x = 1$ *and* $y = 2$ *to approximate* $\sqrt{2}$. *Compare the results with table 1.1. Also run the code with* $x = 1$ *and* $y = \frac{1}{2}$ *and compare the results with the previous case. Explain what you find.*

Exercise 1.3 *Show that the maximum relative error in Heron's algorithm for approximating* $\sqrt{y}$ *for* $y \in [1/M, M]$, *for a fixed number of iterations and starting with* $x_0 = 1$, *occurs at the ends of the interval:* $y = 1/M$ *and* $y = M$. *(Hint: consider (1.10) and (1.14) and show that the function*

$$\phi(x) = \tfrac{1}{2}(1 + x)^{-1}x^2 \tag{1.34}$$

plays a role in each. Show that ϕ *is increasing on the interval* $[0, \infty[$.*)*

Exercise 1.4 *It is sometimes easier to demonstrate the relative accuracy of an approximation* $\hat{x}$ *to* x *by showing that*

$$|x - \hat{x}| \leq \epsilon' |\hat{x}| \tag{1.35}$$

instead of verifying (1.5) directly. Show that if (1.35) holds, then (1.5) holds with $\epsilon = \epsilon'/(1 - \epsilon')$.

Exercise 1.5 *There is a simple generalization to Heron's algorithm for finding* k*th roots as follows:*

$$x \leftarrow \frac{1}{k}((k-1)x + y/x^{k-1}). \tag{1.36}$$

Show that, if this converges, it converges to a solution of $x^k = y$. *Examine the speed of convergence both computationally and by estimating the error algebraically.*

Exercise 1.6 *Show that the error in Heron's algorithm for approximating* $\sqrt{y}$ *satisfies*

$$\frac{x_n - \sqrt{y}}{x_n + \sqrt{y}} = \left(\frac{x_0 - \sqrt{y}}{x_0 + \sqrt{y}}\right)^{2^n} \tag{1.37}$$

for $n \geq 1$. *Note that the denominator on the left-hand side of (1.37) converges rapidly to* $2\sqrt{y}$.

Exercise 1.7 *We have implicitly been assuming that we were attempting to compute a positive square-root with Heron's algorithm, and thus we always started with a positive initial guess. If we give zero as an initial guess, there is immediate failure because of division by zero. But what happens if we start with a negative initial guess? (Hint: there are usually two roots to* $x^2 = y$, *one of which is negative.)*

Exercise 1.8 *Consider the iteration*

$$x \leftarrow 2x - yx^2 \tag{1.38}$$

and show that, if it converges, it converges to $x = 1/y$. *Note that the algorithm does not require a division. Determine the range of starting values* x_0 *for which this will converge. What sort of scaling (cf. section 1.1.2) would be appropriate for computing* $1/y$ *before starting the iteration?*

Exercise 1.9 *Consider the iteration*

$$x \leftarrow \tfrac{3}{2}x - \tfrac{1}{2}yx^3 \tag{1.39}$$

and show that, if this converges, it converges to $x = 1/\sqrt{y}$. *Note that this algorithm does not require a division. The computation of* $1/\sqrt{y}$ *appears in the Cholesky algorithm in (4.12).*

Exercise 1.10 *Suppose that* $a + by$ *is the best linear approximation to* $\sqrt{y}$ *in terms of relative error on* $[\frac{1}{2}, 2]$. *Prove that the error expression* e_{ab} *has to be negative at its minimum. (Hint: if not, you can always decrease* a *to make* $e_{ab}(2)$ *and* $e_{ab}(\frac{1}{2})$ *smaller without increasing the maximum value of* $|e_{ab}|$.*)*

Exercise 1.11 *Suppose that* $a + by$ *is the best linear approximation to* $\sqrt{y}$ *in terms of relative error on* $[\frac{1}{2}, 2]$. *Prove that* $a = b$.

Exercise 1.12 *Suppose that* $a + ay$ *is the best linear approximation to* $\sqrt{y}$ *in terms of relative error on* $[\frac{1}{2}, 2]$. *Prove that the error expression*

$$e_{aa}(1) = -e_{aa}(2). \tag{1.40}$$

(Hint: if not, you can always decrease a *to make* $e_{aa}(2)$ *and* $e_{aa}(\frac{1}{2})$ *smaller without increasing the maximum value of* $|e_{ab}|$.*)*

Exercise 1.13 *Consider the effect of the best starting value of* a *in (1.25) on the Heron algorithm. How many iterations are required to get* 16 *digits of accuracy? And to obtain* 32 *digits of accuracy?*

Exercise 1.14 *Change the function* `minus` *for computing* $\tilde{x}_-$ *and the function* `plusinv` *for computing* $\hat{x}_-$ *to functions for computing* $\tilde{x}_+$ *(call that function* `plus`*) and* $\hat{x}_+$ *(call that function* `minusinv`*). Use the* `check` *function to see where they work well and where they fail. Compare that with the corresponding behavior for* `minus` *and* `plusinv`*.*

Exercise 1.15 *The iteration (1.28) can be implemented via the function*

```
function y =sosimpl(x,a)
y=x+x^2-a;
```

Use this to verify that `sosimpl(1,1)` *is indeed* 1*, but if we start with*

```
x=1.000000000001
```

and then repeatedly apply `x=sosimpl(x,1)`*, the result ultimately diverges.*

1.7 SOLUTIONS

Solution of Exercise 1.3. The function $\phi(x) = \frac{1}{2}(1+x)^{-1}x^2$ is increasing on the interval $[0, \infty[$ since

$$\phi'(x) = \tfrac{1}{2}\frac{2x(1+x) - x^2}{(1+x)^2} = \tfrac{1}{2}\frac{2x + x^2}{(1+x)^2} > 0 \tag{1.41}$$

for $x > 0$. The expression (1.10) says that

$$\hat{e}_{n+1} = \phi(\hat{e}_n), \tag{1.42}$$

and (1.14) says that

$$\hat{e}_1 = \phi(x - 1). \tag{1.43}$$

Thus

$$\hat{e}_2 = \phi(\phi(x - 1)). \tag{1.44}$$

By induction, define

$$\phi^{[n+1]}(t) = \phi(\phi^{[n]}(t)), \tag{1.45}$$

where $\phi^{[1]}(t) = \phi(t)$ for all t. Then, by induction,

$$\hat{e}_n = \phi^{[n]}(x - 1) \tag{1.46}$$

for all $n \geq 1$. Since the composition of increasing functions is increasing, each $\phi^{[n]}$ is increasing, by induction. Thus $\hat{e}_n$ is maximized when x is maximized, at least for $x > 1$. Note that

$$\phi(x - 1) = \phi((1/x) - 1), \tag{1.47}$$

so we may also write

$$\hat{e}_n = \phi^{[n]}((1/x) - 1). \tag{1.48}$$

Thus the error is symmetric via the relation

$$\hat{e}_n(x) = \hat{e}_n(1/x). \tag{1.49}$$

Thus the maximal error on an interval $[1/M, M]$ occurs simultaneously at $1/M$ and M.

Solution of Exercise 1.6. Define $d_n = x_n + x$. Then (1.37) in exercise 1.6 is equivalent to the statement that

$$\frac{e_n}{d_n} = \left(\frac{e_0}{d_0}\right)^{2^n}. \tag{1.50}$$

Thus we compute

$$\begin{aligned} d_{n+1} &= x_{n+1} + x = \tfrac{1}{2}(x_n + y/x_n) + \tfrac{1}{2}(x + y/x) = \tfrac{1}{2}(d_n + y/x_n + y/x) \\ &= \tfrac{1}{2}\left(d_n + \frac{y(x + x_n)}{x x_n}\right) = \tfrac{1}{2}\left(d_n + \frac{y d_n}{x x_n}\right) = \tfrac{1}{2}\left(d_n + \frac{x d_n}{x_n}\right) \\ &= \tfrac{1}{2} d_n \left(1 + \frac{x}{x_n}\right) = \tfrac{1}{2} d_n \left(\frac{x_n + x}{x_n}\right) = \tfrac{1}{2}\frac{d_n^2}{x_n}. \end{aligned} \tag{1.51}$$

Recall that (1.8) says that $e_{n+1} = \frac{1}{2}e_n^2/x_n$, so dividing by (1.51) yields

$$\frac{e_{n+1}}{d_{n+1}} = \left(\frac{e_n}{d_n}\right)^2 \tag{1.52}$$

for any $n \geq 0$. A simple induction on n yields (1.50), as required.