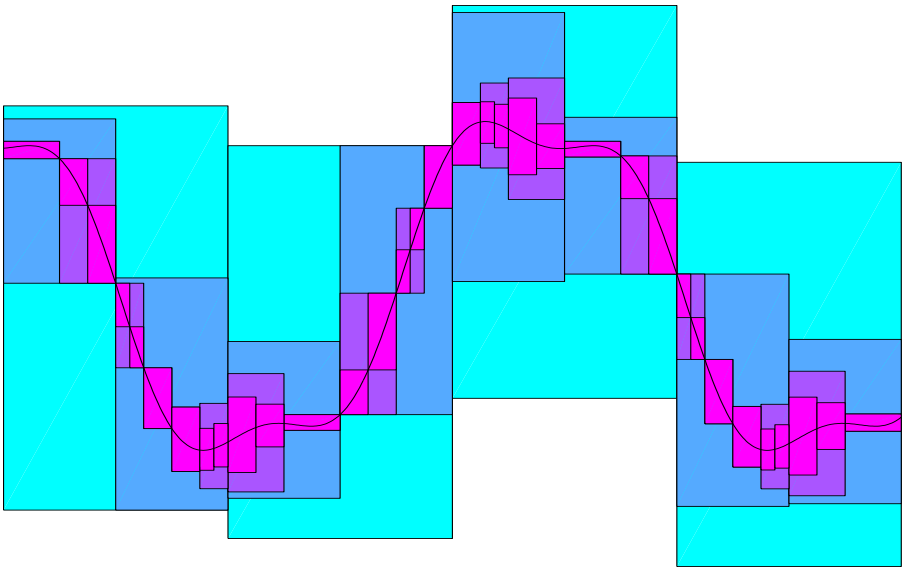
Auto-validating numerical methods

– FMB, Spring '09 –

Warwick Tucker



Warwick Tucker
Department of Mathematics
Uppsala University
Box 480, Uppsala, Sweden
warwick@math.uu.se

Contents

Introduction

Why validated numerics?

Validated numerics is the field aiming at bridging the gap between scientific computing and pure mathematics – between speed and reliability. In many applications, it is of utmost importance to get the correct answer, not an almost correct one. In mathematical research, it is often important to prove that a numerically observed phenomenon has a mathematical counter-part. Validated numerics makes this possible.

One strain of the field aims at pushing the frontiers of computer-aided proofs in mathematical analysis. This area of research deals with problems that cannot be solved by traditional mathematical methods. Typically, such problems have a global component (e.g. a large search space) as well as a non-linear ingredient (the output is not proportional to the input). Such hard problems have traditionally been studied through numerical computations, and therefore our knowledge of these lack the rigour demanded by a formal proof. This type of research aims to bridge the gap between a numerically observed phenomenon, and its mathematical counter-part. This is achieved by developing a means of computing numerically yet with mathematical rigour.

Validated numerics merges pure mathematics with scientific computing in a novel way: instead of computing *approximations* to sought quantities, the aim is to compute *enclosures* of the same. The width of such an enclosure gives a direct quality measurement of the computation, and can be used to adaptively improve the calculations at run-time. This fundamental change of focus results in efficient and extremely fault-tolerant numerical methods, ideal for the systematic study of complex systems. As such, validated numerics will play an instrumental role as computers become increasingly dominant in scientific research, replacing the need for physical experimentation. Indeed, it is the only way to certify that a numerical computation meets required error tolerances. This is e.g. important in control of industrial robots, and vital in the manufacturing of new drugs.

All of these problems share the same type of inaccessibility due to non-linearities affecting the global behaviour of the systems. Neither tools from pure mathematics nor scientific computation alone have been successful in establishing quantitative

information for such complex systems. This emerging field will not only develop the mathematical foundation needed to overcome these obstacles, it will also result in concrete numerical methods able to provide mathematical statements about such systems.

Scope and aim of this book

The main goal of this text is to introduce the reader to the field of auto-validating algorithms by providing a theoretical foundation supplemented with illuminating examples. The target audience is undergraduate or graduate students new to the field. Some knowledge of programming is useful, but not strictly necessary. The restriction to the one-dimensional setting is a conscious one. It allows us to focus exclusively on simple, but yet interesting problems, without too much mathematical framework. The computer excersises are intended to "force" the reader into actually discovering how simple it is to implement the methods, and to solve numerical problems with rigour.

Acknowledgments

I would like to thank all students who have helped me transform my somewhat erratic lecture notes into a more self-contained, polished manuscript. This has been a lengthy process, and without the enthusiasm of the participants of my course on validated numerics, I would not have come this far.

Chapter 1

Computer arithmetic

In this chapter, we give an elementary overview of how (and what type of) numbers are represented, stored and manipulated in a computer. This will provide some insight as to why some floating point computations produce grossly incorrect results. This topic is covered much more extensively in e.g. [Hi96], [Ov01], and [Wi63].

1.1 Positional systems

Our everyday decimal number system is a *positional system* in base 10. Since computer arithmetic is often built on positional systems in other bases (e.g. 2 or 16)[1], we will begin this section by recalling how real numbers are represented in a positional system with an arbitrary integer base $\beta \geq 2$. Setting aside practical restrictions, such as the finite storage capabilities of a computer, any real number can be expressed as an infinite string

$$(-1)^\sigma (b_n b_{n-1} \dots b_0 . b_{-1} b_{-2} \dots)_\beta, \tag{1.1}$$

where $b_n, b_{n-1}, \dots$ are integers in the range $[0, \beta - 1]$, and $\sigma \in \{0, 1\}$ provides the sign of the number. The real number corresponding to (1.1) is

$$\begin{aligned} x &= (-1)^\sigma \sum_{i=-\infty}^{n} b_i \beta^i \\ &= (-1)^\sigma (b_n \beta^n + b_{n-1}\beta^{n-1} + \dots + b_0 + b_{-1}\beta^{-1} + b_{-2}\beta^{-2} + \dots). \end{aligned}$$

If the number ends in an infinite number of consecutive 0:s we omit them in the expression (1.1). Thus we write $(12.25)_{10}$ instead of $(12.25000\dots)_{10}$. Also, we omit any 0:s preceding the integer part $(-1)^\sigma (b_n b_{n-1} \dots b_0)_\beta$. Thus we write $(12.25)_{10}$

[1] Of course, some exceptions do exist: most calculators still use base 10; the Russian computer *Setun* used base 3, whereas the American *Maniac II* used base $65536 = 16^4$!

instead of $(0012.25)_{10}$, and $(0.0025)_{10}$ instead of $(000.0025)_{10}$. Allowing for either leading or trailing extra 0:s is called *padding*, and is not common practice since it leads to redundancies in the representation.

Even without padding, the positional system is slightly flawed. No matter what base we choose, there are still real numbers that do not have a unique representation. For example, the decimal number $(12.2549999\ldots)_{10}$ is equal to $(12.255)_{10}$, and the binary number $(100.01101111\ldots)_2$ is equal to $(100.0111)_2$. This redundancy, however, can be avoided if we add the requirement that $0 \leq b_i \leq \beta - 2$ for infinitely many i.

Exercise 1.1.1 *Prove that any real number $x \neq 0$ has a unique representation (1.1) in a positional system (allowing no padding) with integer base $\beta \geq 2$ under the two conditions (a) $0 \leq b_i \leq \beta - 1$ for all i, and (b) $0 \leq b_i \leq \beta - 2$ for infinitely many i.*

Exercise 1.1.2 *What is the correct way to represent zero in a positional system allowing no padding?*

1.2 Floating point numbers

When expressing a real number on the form (1.1), the placement of the decimal[2] point is crucial. The *floating point* number system provides a more convenient way to represent real numbers. A floating point number is a real number on the form

$$x = (-1)^\sigma m \times \beta^e, \tag{1.2}$$

where $(-1)^\sigma$ is the sign of x, m is called the *mantissa*[3], and e is called the *exponent* of x. Writing numbers in floating point notation frees us from the burden of keeping track of the decimal point: it always follows the first digit of the mantissa. It is customary to write the mantissa as

$$m = (b_0.b_1b_2\ldots)_\beta$$

where, compared to the previous section, the indexing of the b_i has opposite sign. As this is the standard notation, we will adopt this practice in what follows. Thus we may define the set of floating point numbers in base β as:

$$\mathbb{F}_\beta = \{(-1)^\sigma m \times \beta^e \colon m = (b_0.b_1b_2\ldots)_\beta\},$$

where, as before, we request that β is an integer no less than 2, and that $0 \leq b_i \leq \beta - 1$ for all i, and $0 \leq b_i \leq \beta - 2$ for infinitely many i. The exponent e may be any integer.

[2]Being picky, the expression "*base* point" or "*radix* point" is more appropriate, unless $\beta = 10$.

[3]The mantissa is sometimes referred to as the *significand* or, rather incorrectly, as the *fractional part* of the floating point number.

Expressing real numbers in floating point form introduces a new type of redundancy. For example, the base-10 number 123 can be expressed as $(1.23)_{10} \times 10^2$, $(0.123)_{10} \times 10^3$, $(0.0123)_{10} \times 10^4$, and so on. In order to have unique representations for non-zero real numbers, we demand that the leading digit b_0 be non-zero, except for the special case $x = 0$. Floating point numbers satisfying this additional requirement are said to be *normal* or *normalized*[4].

Exercise 1.2.1 *Show that a non-zero floating point number is normal iff its associated exponent e is chosen minimal.*

So far, we have simply toyed with different representations of the real numbers $\mathbb{R}$. As this set is uncountably infinite, whereas a machine can only store a finite amount of information, more drastic means are called for: we must introduce a new, much smaller, set of numbers designed to fit into a computer, and which at the same time approximate the real numbers in some well-defined sense.

As a first step toward this goal, we restrict the number of digits representing the mantissa. This yields the set

$$\mathbb{F}_{\beta,p} = \{x \in \mathbb{F}_\beta \colon m = (b_0.b_1b_2 \ldots b_{p-1})_\beta\}.$$

The number p is called the *precision* of the floating point system. It is a nice exercise to show that, although $\mathbb{F}_{\beta,p}$ is a much smaller set than $\mathbb{F}_\beta$, it is countably infinite. This means that even $\mathbb{F}_{\beta,p}$ is too large for our needs. Note, however, that the restriction $0 \leq b_i \leq \beta - 2$ for infinitely many i becomes void in finite precision.

A *finite* set of floating point numbers can be formed by imposing a fixed precision, as well as bounds on the admissible exponents. Such a set is specified by four integers: the base β, the precision p, and the minimal and maximal exponents $\check{e}$ and $\hat{e}$, respectively. Given these quantities we can define parameterized sets of *computer representable* floating point numbers:

$$\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} = \{x \in \mathbb{F}_{\beta,p} \colon \check{e} \leq e \leq \hat{e}\}.$$

Exercise 1.2.2 *Show that $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ is finite, whereas $\mathbb{F}_{\beta,p}$ is countably infinite, and $\mathbb{F}_\beta$ is uncountably infinite, with*

$$\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} \subset \mathbb{F}_{\beta,p} \subset \mathbb{F}_\beta.$$

Exercise 1.2.3 *How many normal numbers belong to the set $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$?*

Using a base other than 10 forces us to have to rethink which numbers have a finite representation. This can cause some confusion to the novice programmer.

[4]This should not be confused with the number-theoretic notion of a normal number. There a number is normal to base β if every sequence of n consecutive digits in its β-expansion appears with limiting probability β^{-n}.

Example 1.2.4 *With $\beta = 2$ and $p < \infty$, the number $1/10$ is not exactly representable in $\mathbb{F}_{\beta,p}$. This can be seen by noting that*

$$\sum_{k=1}^{\infty} \left(2^{-4k} + 2^{-(4k+1)}\right) = \frac{3}{2}\left(\frac{1}{1-2^{-4}} - 1\right) = \frac{1}{10}.$$

Interpreting the first sum as a binary representation, we have

$$1/10 = (0.00011001100110011\ldots)_2 = (1.1001100110011\ldots)_2 \times 2^{-4}.$$

Since this is a non-terminating binary number, it has no exact representation in $\mathbb{F}_{2,p}$, regardless of the choice of precision p.

This example may come as a surprise to all who use the popular step-sizes 0.1 or 0.001 in, say, numerical integration methods. Most computers use $\beta = 2$ in their internal floating point representation, which means that on these computers $1000 \times 0.001 \neq 1$. This simple fact can have devastating consequences for e.g. very long integration procedures. More suitable step-sizes would be $2^{-3} = 0.125$ and $2^{-10} = 0.0009765625$, which are exactly representable in base two. Example 1.4.2 further illustrates how sensitive numerical computations can be to these small conversion errors.

1.2.1 Subnormal numbers

As mentioned earlier, without the normalizing restriction $b_0 \neq 0$, a non-zero real number may have several representations in the set $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$. (Note, however, that most real numbers no longer have *any* representation in this finite set.) We already remarked that these redundancies may be avoided by normalization, i.e., by demanding that all non-zero floating point numbers have a non-zero leading digit. To illustrate the concept of normalized numbers, we will study a small toy system of floating point numbers, illustrated in Figure 1.1.

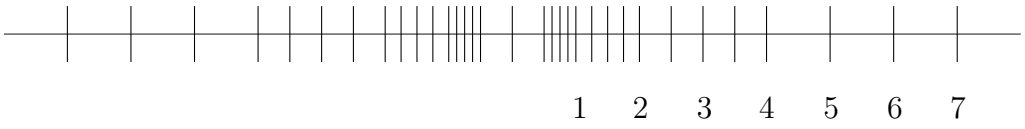


Figure 1.1: The normal numbers of $\mathbb{F}_{2,3}^{-1,2}$.

It is clear that the floating point numbers are not uniformly distributed: the positive numbers are given by $\{0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7\}$. Thus the intermediate distances between consecutive numbers range within the set $\{0.125, 0.25, 0.5, 1\}$. In Section 1.3, we will explain why this type of non-uniform spacing is a good idea. We will also describe how to deal with real numbers having modulus greater than the largest positive normal number N_{max}^n, which in our toy system is $(1.11)_2 \times 2^2 = 7$.

Note that the smallest positive normal number in $\mathbb{F}_{2,3}^{-1,2}$ is $N_{min}^n = (1.00)_2 \times 2^{-1} = 0.5$, which leaves an undesired gap centered around the origin. Not only does this lead to a huge loss of accuracy when approximating numbers of small magnitude, it also leads to the violation of some of our most valued mathematical laws, see Exercise 1.4.5. A way to work around these problems is to allow for some numbers that are not normal.

A non-zero floating point number in $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ with $b_0 = 0$ and $e = \check{e}$, it is said to be *subnormal* (or denormalized). Subnormal numbers allow for a *gradual underflow* to zero (compare Figures 1.2 and 1.1). Extending the set of admissible floating point numbers to include the subnormal numbers still preserves uniqueness of representation, although the use of these additional numbers comes with some penalties, as we shall shortly see. For our toy system at hand, the positive subnormal numbers are $\{0.125, 0.25, 0.375\}$.

Figure 1.2 illustrates the normal and subnormal numbers of $\mathbb{F}_{2,3}^{-1,2}$:

Figure 1.2: The normal and subnormal numbers of $\mathbb{F}_{2,3}^{-1,2}$.

The difference between these two sets is striking: the smallest positive normal number N_{min}^n is $(1.00)_2 \times 2^{-1} = 0.5$, whereas the smallest positive subnormal number N_{min}^s is $(0.01)_2 \times 2^{-1} = 0.125$. The largest subnormal number N_{max}^s is $(0.11)_2 \times 2^{-1} = 0.375$. Geometrically, introducing subnormal numbers corresponds to filling the gap centered around the origin with evenly spaced numbers. The spacing should be the same as between the two smallest positive normal numbers.

From now on, when we refer to the set $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$, we mean the set of normal *and* subnormal numbers. We will use $\mathbb{F}$ to denote any set of type $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ or $\mathbb{F}_{\beta,p}$; when needed, the exact parameters of the set in question will be provided. A real number x with $N_{min}^n \leq |x| \leq N_{max}^n$ is said to be in the *normalized range* of the associated floating point system.

Exercise 1.2.5 *Prove that the non-zero normal and subnormal numbers of* $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ *have unique representations.*

Exercise 1.2.6 *How many positive subnormal numbers are there in the set* $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$*?*

Exercise 1.2.7 *Derive formulas for* N_{min}^s*,* N_{max}^s*,* N_{min}^n*, and* N_{max}^n *for a general set* $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$*.*

We conclude this section by remarking that, although the floating point numbers are not uniformly spaced, for $\check{e} \leq m, n < \hat{e}$, the sets $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} \cap [\beta^m, \beta^{m+1}]$ and $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} \cap [\beta^n, \beta^{n+1}]$

have the same cardinality. This is apparent in Figures 1.2 and 1.1. We also note that any floating point system $\mathbb{F}$ is symmetric with respect to the origin: $x \in \mathbb{F} \Leftrightarrow -x \in \mathbb{F}$.

Exercise 1.2.8 *Compute the number of elements of the set* $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}} \cap [\beta^n, \beta^{n+1})$*, provided that* $\check{e} \leq n < \hat{e}$.

1.3 Rounding

We have now reached the stage where we we have succeeded to condense the uncountable set of real numbers $\mathbb{R}$ into a finite set of floating point numbers $\mathbb{F}$. Almost all commercial computers use a set like $\mathbb{F}$, with some minor additions, to approximate the real numbers. In order to make computing feasible over $\mathbb{F}$, we must find a means to associate any real number $x \in \mathbb{R}$ to a member y of $\mathbb{F}$. Such an association is called *rounding*, and defines a map from $\mathbb{R}$ onto $\mathbb{F}$. Obviously, we cannot make the map invertible, but we would like to make it as close as possible to a homeomorphism.

Before defining such a mapping, we will extend both the domain and range into the sets $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$ and $\mathbb{F}^* = \mathbb{F} \cup \{-\infty, +\infty\}$, respectively. This provides an elegant means for representing real numbers that are too large in absolute value to fit into $\mathbb{F}$. In the actual implementation, the symbols $\{-\infty, +\infty\}$ are specially coded, and do not have a valid representation such as (1.2).

Following the excellent treatise on computer arithmetic [KM81], a rounding operation $\bigcirc\colon \mathbb{R}^* \to \mathbb{F}^*$ should have the following properties:

(R1) $x \in \mathbb{F}^* \Rightarrow \bigcirc(x) = x$,

(R2) $x, y \in \mathbb{R}^*$ and $x \leq y \Rightarrow \bigcirc(x) \leq \bigcirc(y)$.

Property (R1) simply states that all members of $\mathbb{F}^*$ are fixed under $\bigcirc$. Clearly an already representable number does not require any further rounding. Property (R2) states that the rounding is monotone. Indeed, it would be very difficult to interpret the meaning of any numerical procedure without this property. Combining (R1) and (R2), one can easily prove that the rounding $\bigcirc$ is of *maximum quality*, i.e., the interior of the interval spanned by x and $\bigcirc(x)$ contains no points of $\mathbb{F}^*$.

Lemma 1.3.1 *Let* $x \in \mathbb{R}^*$*. If* $\bigcirc\colon \mathbb{R}^* \to \mathbb{F}^*$ *satisfies both (R1) and (R2), then the interval spanned by* x *and* $\bigcirc(x)$ *contains no points of* $\mathbb{F}^*$ *in its interior.*

Proof: The claim is trivially true if $x = \bigcirc(x)$ (since then the interior is empty), so assume that $x \neq \bigcirc(x)$. Without loss of generality we may assume that $x < \bigcirc(x)$. Now suppose that the claim is false, i.e., there exists an element $y \in \mathbb{F}^*$ with $x <$

$y < \bigcirc(x)$. Since, by (R1), we have $\bigcirc(y) = y$, we must, by (R2), have $\bigcirc(x) \leq y$. This gives the desired contradiction. □

We will now describe four rounding modes that are available on most commercial computers

1.3.1 Round to zero

The simplest rounding operation to implement is *round toward zero*, often referred to as *truncation*, which we denote by $\square_z$. We formally define $\square_z \colon \mathbb{R}^* \to \mathbb{F}^*$ by

$$\square_z(x) = \text{sign}(x) \max\{y \in \mathbb{F}^* \colon y \leq |x|\}, \tag{1.3}$$

where $\text{sign}(x)$ is the sign of x. The action of $\square_z$ is illustrated in Figure 1.3. To see how easy this rounding mode is to implement, consider a real number $x = (-1)^\sigma (b_0.b_1 b_2 \ldots)_\beta \times \beta^e$ in $\mathbb{F}_\beta$ to be rounded into $\mathbb{F}_{\beta,p}$. If x satisfies $|x| \leq N^n_{max}$, this is achieved by simply discarding the mantissa digits beyond position $p-1$ (hence the nickname truncation): $\square_z(x) = (-1)^\sigma (b_0.b_1 b_2 \ldots b_{p-1})_\beta \times \beta^e$. Otherwise, x is rounded to $(-1)^\sigma N^n_{max}$.

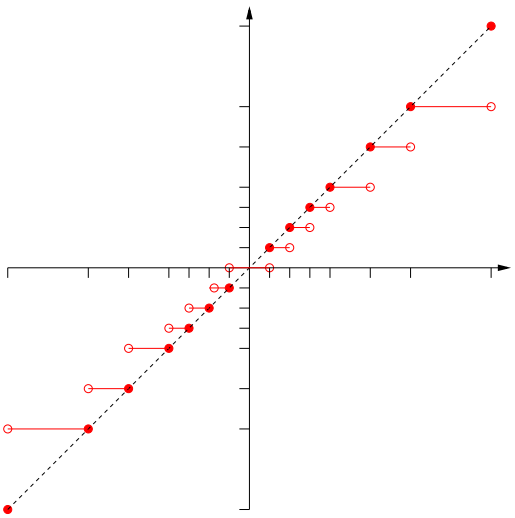


Figure 1.3: Round to zero $\square_z$.

Round toward zero is an *odd*[5] function:

(R3) $\quad x \in \mathbb{R}^* \Rightarrow \bigcirc(-x) = -\bigcirc(x).$

Rule (R3) is also satisfied by the most common rounding mode: *round to nearest*, which we shall return to later.

[5] A function f is said to be *odd* if $f(-x) = -f(x)$, and *even* if $f(-x) = f(x)$. Most functions are neither.

1.3.2 Directed rounding

There are two very useful rounding modes that are said to be *directed*. By this we mean that they satisfy (in addition to (R1) and (R2)) one of the following rules:

$$\text{(R4)} \qquad \text{(a) } x \in \mathbb{R}^* \Rightarrow \bigcirc(x) \leq x \quad \text{ or } \quad \text{(b) } x \in \mathbb{R}^* \Rightarrow \bigcirc(x) \geq x.$$

The rounding mode satisfying (R4a) is called *round toward minus infinity* (or simply *round down*). The rounding mode satisfying (R4b) is called *round toward plus infinity* (or simply *round up*). These rounding modes are denoted $\bigtriangledown(x)$ and $\bigtriangleup(x)$, respectively, and are formally defined by

$$\bigtriangledown(x) = \max\{y \in \mathbb{F}^* \colon y \leq x\} \quad \text{and} \quad \bigtriangleup(x) = \min\{y \in \mathbb{F}^* \colon y \geq x\}. \tag{1.4}$$

The number $\bigtriangledown(x)$, called x *rounded down*, is the largest floating point number less than or equal to x, whereas $\bigtriangleup(x)$, called x *rounded up*, is the smallest floating point number greater than or equal to x. Note that, if $x \in \mathbb{F}^*$, then $\bigtriangledown(x) = x = \bigtriangleup(x)$, whereas if $x \notin \mathbb{F}^*$, we have the enclosure $\bigtriangledown(x) < x < \bigtriangleup(x)$, which is of maximal quality. This means that the interval $[\bigtriangledown(x), \bigtriangleup(x)]$ contains no points of $\mathbb{F}^*$ in its interior (apply Lemma 1.3.1 twice). Also note the anti-symmetry relations:

$$\bigtriangleup(-x) = -\bigtriangledown(x) \quad \text{and} \quad \bigtriangledown(-x) = -\bigtriangleup(x). \tag{1.5}$$

Thus either rounding $\bigtriangledown$ or $\bigtriangleup$ can be completely defined in terms of the other.

Exercise 1.3.2 *Using only (1.4), prove that the rounding modes $\bigtriangledown$ and $\bigtriangleup$ satisfy (R1) and (R2). Also show that $\bigtriangledown$ satisfies (R4a), and that $\bigtriangleup$ satisfies (R4b).*

Exercise 1.3.3 *Show that, in terms of the directed rounding mode $\bigtriangledown$, we have*

$$\square_z(x) = \text{sign}(x) \bigtriangledown (|x|).$$

The relations (1.4) completely define the directed roundings $\bigtriangledown$ and $\bigtriangleup$ as maps from $\mathbb{R}^*$ to $\mathbb{F}^*$, see Figure 1.4.

1.3.3 Round to nearest (even)

Note that all previously defined rounding modes map the interior any interval spanned by two consecutive floating point numbers onto a single point in $\mathbb{F}^*$. This means that the error made when rounding a single real number x could be as large as the length of the interval $[\bigtriangledown(x), \bigtriangleup(x)]$ enclosing it. A more accurate family of rounding modes is called *round to nearest*.

For an element of $x \in \mathbb{R}^*$, we can construct an enclosure of x in $\mathbb{F}^*$:

$$\bigtriangledown(x) \leq x \leq \bigtriangleup(x).$$

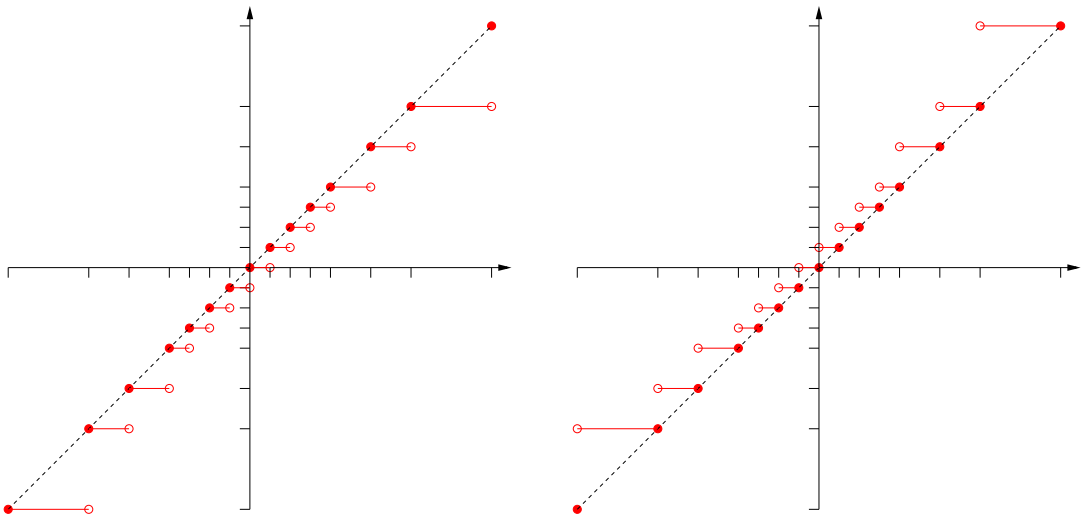


Figure 1.4: Directed roundings: (a) round down $\bigtriangledown$; (b) round up $\bigtriangleup$.

If also $|x| \leq N^n_{max}$, we let $\mu = \frac{1}{2}(\bigtriangleup(x) + \bigtriangledown(x))$ denote the midpoint of this interval. If $|x| > N^n_{max}$, we let $\mu = \text{sign}(x)N^n_{max}$. Rounding to nearest returns $\bigtriangledown(x)$ if $x < \mu$, and $\bigtriangleup(x)$ if $x > \mu$. In the rare case $x = \mu$, there is a tie. The different variants of rounding to nearest are distinguished according to how they resolve this tie.

One easy way to break the tie, is to simply round up for positive ties, and down for negative ones. This rounding mode is called *round to nearest*, and is defined by

$$\begin{aligned} x > 0 \Rightarrow \square_n(x) &= \begin{cases} \bigtriangledown(x), & \text{if } x \in [\bigtriangledown(x), \mu), \\ \bigtriangleup(x), & \text{if } x \in [\mu, \bigtriangleup(x)], \end{cases} \\ x < 0 \Rightarrow \square_n(x) &= -\square_n(-x). \end{aligned} \tag{1.6}$$

Although easy to describe, this rounding mode has the slight disadvantage of being *biased*: the rounding errors are not evenly distributed around zero. Indeed, if x is positive, then $\square_n$ has a higher probability of rounding x downward, and vice-versa, see Figure 1.5a.

An unbiased way to break the tie gives rise to a rounding mode called *round to nearest even*, which we simply denote by $\square$. This is the default rounding mode on almost all commercial computers. In order to define this rounding operation, we will assume that the mantissas of $\bigtriangledown(x)$ and $\bigtriangleup(x)$ are given by

$$(a_0.a_1a_2\ldots a_{p-1})_\beta \quad \text{and} \quad (b_0.b_1b_2\ldots b_{p-1})_\beta,$$

respectively. Note that, by Lemma 1.3.1, if $x \notin \mathbb{F}^*$ then *exactly* one of the integers a_{p-1} and b_{p-1} is even. We define the *round to nearest even* scheme by

$$\begin{aligned} x > 0 \Rightarrow \square(x) &= \begin{cases} \bigtriangledown(x), & \text{if } x \in [\bigtriangledown(x), \mu), \text{ or if } x = \mu \text{ and } a_{p-1} \text{ is even}, \\ \bigtriangleup(x), & \text{if } x \in (\mu, \bigtriangleup(x)], \text{ or if } x = \mu \text{ and } b_{p-1} \text{ is even}, \end{cases} \\ x < 0 \Rightarrow \square(x) &= -\square(-x). \end{aligned} \tag{1.7}$$

Note that this definition evens out the probability of rounding upward or downward; the rounding is *unbiased*, see Figure 1.5b.

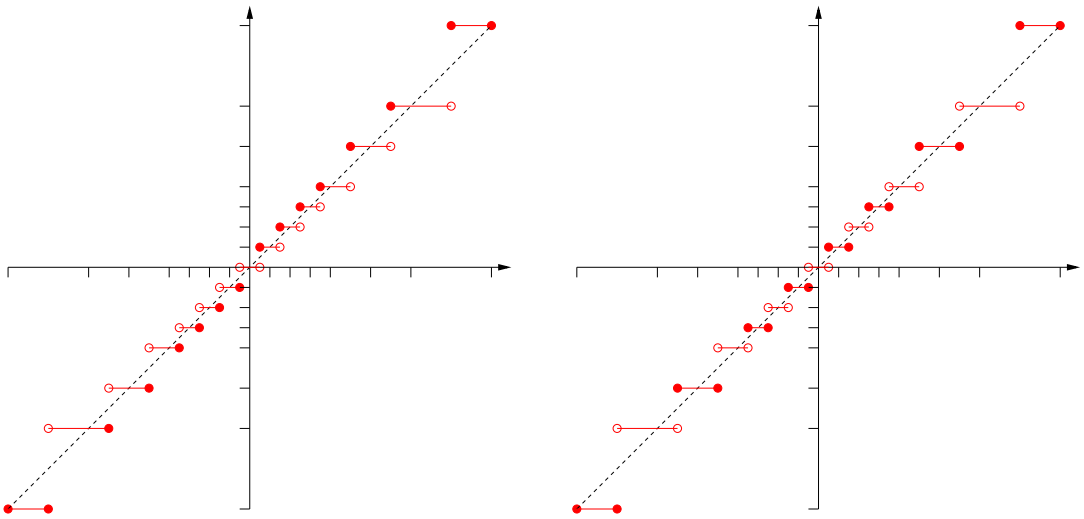


Figure 1.5: Round to nearest: (a) biased $\square_n$; (b) unbiased $\square$.

When rounding a real number x of very large magnitude ($|x| > N^n_{max}$) it should be pointed out that, despite the name *round to nearest*, x is actually rounded to $\text{sign}(x)\infty$, while $\text{sign}(x)N^n_{max}$ actually is the nearest element in $\mathbb{F}^*$.

As a final remark, it is clear that round to nearest *odd* can be defined in a completely analogous manner. One may then wonder whether the choice between rounding to nearest even or to nearest odd is relevant. The answer is, somewhat surprisingly, Yes! This is demonstrated in the following example.

Example 1.3.4 *Consider the following scenario: let $\beta = 10$, $x = 0.45$, and suppose that we want to round x to two digits, after which we continue to round the result to one digit. Using round to nearest even produces $0.45 \to 0.4 \to 0 = \tilde{x}_e$, whereas round to nearest odd produces $0.45 \to 0.5 \to 1 = \tilde{x}_o$, which is not the nearest single-digit number seeing that $|x - \tilde{x}_e| = 0.45 < 0.55 = |x - \tilde{x}_o|$.*

Now, suppose that $\beta = 4$, $x = (0.22)_4$, and suppose once again that we want to round x to two digits, after which we continue to round the result to one digit. Using round to nearest even produces $(0.22)_4 \to (0.2)_4 \to (0)_4 = \tilde{x}_e$, whereas round to nearest odd produces $(0.22)_4 \to (0.3)_4 \to (1)_4 = \tilde{x}_o$, which now is the nearest single-digit number seeing that $|x - \tilde{x}_e| = (0.22)_4 = 5/8 > 3/8 = (0.21)_4 = |x - \tilde{x}_o|$.

This example illustrates the fact that, when using an even base β, with $\beta/2$ *odd*, the best choice is round to nearest *even*. For an even base β, with $\beta/2$ *even*, however, the best choice is round to nearest *odd*. If the base itself β is odd, we can never have a tie (at least not in finite precision), so the choice of rounding never arises.

In particular, this means that for the popular choices $\beta = 10$ or 2, we should use round to nearest *even*, whereas for $\beta = 16$, round to nearest *odd* is the superior choice.

1.3.4 Rounding errors

In the normalized range, the error produced when rounding a real number to a floating point system can be bounded in terms of the base β and precision p. We have the following bounds on the relative and absolute rounding errors:

Theorem 1.3.5 *If x is a real number in the normalized range of $\mathbb{F} = \mathbb{F}_{\beta,p}$, then the relative error caused by rounding is bounded by $\varepsilon_M = \beta^{-(p-1)}$:*

$$\left| \frac{x - \bigcirc(x)}{x} \right| < \varepsilon_M.$$

Equivalently, the corresponding absolute error is bounded by $|x|\varepsilon_M$:

$$|x - \bigcirc(x)| < |x|\varepsilon_M.$$

The number $\varepsilon_M = \beta^{-(p-1)}$ is called the *machine epsilon*, and is a very useful quantity in numerical error analysis. It is the distance between 1.0 and the next larger floating point number. We will encounter ε_M on several occasions throughout this text.

Proof: Without loss of generality, we may assume that x is positive. If x happens to be a member of $\mathbb{F}$, then there is no rounding error, and the claim follows trivially. Thus we only need to consider the case $x \notin \mathbb{F}$. Since x is in the normalized range, it has a representation $x = (b_0.b_1b_2 \ldots b_{p-1}b_p \ldots)_\beta \times \beta^e$, where $b_0 \neq 0$. Using the fact that $x \notin \mathbb{F}$, it follows that its nearest neighbours in $\mathbb{F}$, $\bigtriangledown(x) = (b_0.b_1b_2 \ldots b_{p-1})_\beta \times \beta^e$ and $\bigtriangleup(x) = [(b_0.b_1b_2 \ldots b_{p-1})_\beta + \beta^{-(p-1)}] \times \beta^e$ are separated by the distance given by $\bigtriangleup(x) - \bigtriangledown(x) = \beta^{-(p-1)} \times \beta^e = \varepsilon_M\beta^e$, and so $|x - \bigcirc(x)| < \varepsilon_M\beta^e$. Since x is normalized, we have that $x \geq 1 \times \beta^e$, so $|x - \bigcirc(x)| < x\varepsilon_M$. This gives the absolute error bound from which the relative bound follows immediately. Negative numbers are treated completely analogously. □

Note that in the case of rounding to nearest, the bounds of Theorem 1.3.5 can be decreased by a factor 0.5.

Exercise 1.3.6 *Show that the spacing between two adjacent floating point numbers x and y in the normalized range is bounded between $|x|\varepsilon_M/\beta$ and $|x|\varepsilon_M$.*

Example 1.3.7 *With base $\beta = 2$ and precision $p = 14$, the correctly rounded fraction $1/10$ is represented as:*

$$\bigtriangledown(1/10) = (1.1001100110011)_2 \times 2^{-4} \qquad \bigtriangleup(1/10) = (1.1001100110100)_2 \times 2^{-4}.$$

Thus we have

$$\left|\frac{1/10 - \bigtriangledown(1/10)}{1/10}\right| = \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-4} - (1.1001100110011)_2 \times 2^{-4}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right|$$
$$= \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-20}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right| = 2^{-16}.$$

$$\left|\frac{1/10 - \bigtriangleup(1/10)}{1/10}\right| = \left|\frac{(1.10011001100110011\ldots)_2 \times 2^{-4} - (1.1001100110100)_2 \times 2^{-4}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right|$$
$$= \left|\frac{(0.11001100110011001\ldots)_2 \times 2^{-15} - (1.0000000000000)_2 \times 2^{-15}}{(1.10011001100110011\ldots)_2 \times 2^{-4}}\right|$$
$$< \left|\frac{(011)_2 \times 2^{-17} - (100)_2 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right| = \left|\frac{3 \times 2^{-17} - 4 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right|$$
$$= \left|\frac{-1 \times 2^{-17}}{(1.100)_2 \times 2^{-4}}\right| = \frac{2}{3} \times 2^{-13}.$$

Both relative errors are clearly bounded by $\varepsilon_M = 2^{-13}$.

It is important to keep in mind that Theorem 1.3.5 does *not* hold for subnormal numbers. In this situation, we can no longer use the fact that the leading digit b_0 in the floating point representation of x is non-zero. This prevents us from obtaining the desired bounds. Nevertheless, a simple modification of the proof of Theorem 1.3.5 gives the following error bounds:

Corollary 1.3.8 *If x is a real number such that $|x| = (b_0.b_1b_2\ldots b_{p-1}b_p\ldots)_\beta \times \beta^{\check{e}}$ with $b_i = 0$ for all $0 \leq i < k \leq p-1$ and $b_k \neq 0$, then the relative error caused by rounding to $\mathbb{F}^{\check{e},\hat{e}}_{\beta,p}$ is bounded by*

$$\left|\frac{x - \bigcirc(x)}{x}\right| < \beta^{-(p-1-k)}.$$

Equivalently, the corresponding absolute error is bounded by

$$|x - \bigcirc(x)| < |x|\beta^{-(p-1-k)}.$$

Thus, rounding in the subnormal range ($N^s_{min} \leq |x| \leq N^s_{max}$) leads to larger relative errors. The alternative, i.e., having no subnormal numbers at all, would result in flushing any number with $|x| < N^n_{min}$ to zero, which is of course even less desirable. Note that the requirement $k \leq p-1$ ensures that $|x|$ is not smaller than the smallest positive subnormal number. Were this the case, the real number $|x|$ could be rounded *down* to zero, yielding a relative error of 1. It could also be rounded *up* to the smallest subnormal number, in which case the relative error could be arbitrarily large.

Example 1.3.9 *In the floating point system* $\mathbb{F}_{2,10}^{-5,5}$*, the correctly rounded real number* 10^{-100} *is represented as:*

$$\bigtriangledown(10^{-100}) = (0.000000000)_2 \times 2^{-5} \qquad \bigtriangleup(10^{-100}) = (0.000000001)_2 \times 2^{-5}.$$

Thus we have the relative error bounds:

$$\left|\frac{10^{-100} - \bigtriangledown(10^{-100})}{10^{-100}}\right| = \left|\frac{10^{-100} - (0.000000000)_2 \times 2^{-5}}{10^{-100}}\right| = 1.$$

$$\begin{aligned}\left|\frac{10^{-100} - \bigtriangleup(10^{-100})}{10^{-100}}\right| &= \left|\frac{10^{-100} - (0.000000001)_2 \times 2^{-5}}{10^{-100}}\right| \\ &= \left|\frac{10^{-100} - 2^{-14}}{10^{-100}}\right| > \frac{10^{-5}}{10^{-100}} = 10^{95}.\end{aligned}$$

1.4 Floating point arithmetic

The main mathematical concern about computing over a set of floating point numbers (i.e. a set $\mathbb{F}$ of type $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$ or $\mathbb{F}_{\beta,p}$) is that it is not arithmetically closed. This means that if we take two floating point numbers $x, y \in \mathbb{F}$, and choose an arithmetic operator $\star \in \{+, -, \times, \div\}$ then, in general, the result will not be exactly representable in the floating point system: $x \star y \notin \mathbb{F}$. This is a property that is *not* shared by the real numbers $\mathbb{R}$, nor even the rationals $\mathbb{Q}$. No matter how high precision we use, this problem remains.

The only way we can define arithmetic on a set of floating point numbers $\mathbb{F}$ then is to associate the exact (in $\mathbb{R}$) outcome of a floating point operation with a representable floating point number. Naturally, this can be achieved by rounding the exact result from $\mathbb{R}$ to $\mathbb{F}$. Given any one of the arithmetic operations $\star \in \{+, -, \times, \div\}$, let $\circledast \in \{\oplus, \ominus, \otimes, \oslash\}$ denote the corresponding operation carried out in $\mathbb{F}$. We say that the floating point arithmetic is of maximum quality if

$$\text{(RG)} \qquad x, y \in \mathbb{F} \text{ and } \star \in \{+, -, \times, \div\} \Rightarrow x \circledast y = \bigcirc(x \star y).$$

Property (RG) states that floating point arithmetic should yield the same result as if though the computation was carried out with infinite precision, after which the exact result is rounded to the appropriate neighbouring floating point. Thus the only error is incurred by the final rounding to $\mathbb{F}$, and consequently, the result of any arithmetic operation has the same quality as the rounding itself. It is true, but not obvious, that this can be practically implemented[6].

[6] In fact, it suffices to use registers having $p + 2$ digits of precision, combined with a so called *sticky bit* to obtain maximum quality in the arithmetic operations, see e.g. [Go91], [Kn98], or [Ko02].

Theorem 1.4.1 *Let $\star$ denote one of the arithmetic operators $+, -, \times, \div$. Then, if x and y are normal floating point numbers with $x \star y \neq 0$, the relative error of the floating point operation is bounded by*

$$\left| \frac{x \star y - x \circledast y}{x \star y} \right| < \varepsilon_M.$$

Equivalently, the corresponding absolute error is bounded by

$$|x \star y - x \circledast y| < |x \star y| \varepsilon_M.$$

It is important to realize that Theorem 1.4.1 is only valid for *one single* floating point operation. All bets are off when several operations are involved. An expression like

$$f(x) = 1 \oslash ((1 \oplus x) \ominus 1)$$

may return a grossly incorrect value when $|x| < \varepsilon_M$, depending on the rounding mode. The following example serves as an illustration to how these types of inaccuracies may seriously affect the outcome of a simple numerical experiment.

Example 1.4.2 *Consider the ternary shift map $f\colon [0,1] \to [0,1]$ defined by $f(x) = 3x \bmod 1$. This map has a period-4 cycle $\frac{1}{10} \to \frac{3}{10} \to \frac{9}{10} \to \frac{7}{10} \to \frac{1}{10}$. Starting a numerical iteration (over $\mathbb{F}_{2,53}$) at $x_0 = \frac{1}{10}$, however, produces the following orbit:*

$$\begin{aligned}
x(0) &= 0.10000000000000001 & x(1) &= 0.30000000000000004 \\
x(2) &= 0.90000000000000013 & x(3) &= 0.70000000000000018 \\
x(4) &= 0.10000000000000053 & x(5) &= 0.3000000000000016 \\
&\vdots \\
x(47) &= 0.15273362128584456 & x(48) &= 0.45820086385753367 \\
x(49) &= 0.37460259157260101 & x(50) &= 0.12380777471780302 \\
x(51) &= 0.37142332415340906 & x(52) &= 0.11426997246022719
\end{aligned}$$

After less than 50 iterates, there is no sign of the periodic orbit! The reason is that the function f is expanding, i.e., its derivate (when defined) is greater than one everywhere. As a consequence, even very small initial errors will eventually be grossly magnified, and completely saturate the computed orbit. This intrinsic property of expanding maps makes the numerical study of chaotic dynamical systems a very challenging topic.

Another consequence of computing with finite precision is that many basic mathematical laws no longer hold. For example, addition and multiplication are no longer associative.

Example 1.4.3 *Consider the numbers* $x = 1.234 \times 10^4$, $y = -1.235 \times 10^4$, *and* $z = 1.002 \times 10^1$, *all belonging to* $\mathbb{F}_{10,4}^{-9,9}$. *Rounding to nearest even, we have*

$$(x \oplus y) \oplus z = -1.000 \times 10^1 \oplus 1.002 \times 10^1 = 2.000 \times 10^{-2},$$

whereas

$$x \oplus (y \oplus z) = 1.234 \times 10^4 \ominus 1.234 \times 10^4 = 0.000 \times 10^{-9}.$$

The first result is exact, while the second suffers from inaccuracies caused by a lack of precision.

Exercise 1.4.4 *How many pairs of floating point numbers can be exactly added in the set* $\mathbb{F}_{2,3}^{-1,2}$? *How many pairs can not? What about the general case* $\mathbb{F}_{\beta,p}^{\check{e},\hat{e}}$?

Exercise 1.4.5 *Assuming the property (RG), show that the following statements always hold true:*

(1) $x \in \mathbb{F} \Rightarrow 1 \otimes x = x$;

(2) $x \in \mathbb{F} \setminus \{-\infty, 0, +\infty\} \Rightarrow x \oslash x = 1$;

(3) $x \in \mathbb{F}$ *(with* $\beta = 2$*)* $\Rightarrow 0.5 \otimes x = x \oslash 2$.

(4) $x, y \in \mathbb{F} \Rightarrow (x \ominus y = 0) \Rightarrow (x = y)$.

Also show that statement (4) is false if $\mathbb{F}$ *has no subnormal numbers.*

In view of (RG), we can give a computational definition of the machine epsilon. Even if the base and precision of the underlying floating point system are unknown, this definition allows for a direct computation of ε_M.

Definition 1.4.6 *In a floating point system with base* β *and precision* p, *we call the number* $\varepsilon_M = \beta^{-(p-1)}$ *the machine epsilon. It is the smallest positive floating point number* x *that satisfies* $1 < 1 \oplus x$ *when rounding down, i.e.,* $\varepsilon_M = \min\{x \in \mathbb{F} \colon 1 < \bigtriangledown(1 \oplus x)\}$.

Note that the condition $1 < \bigtriangledown(1 \oplus x)$ is very different from the *seemingly* equivalent condition $0 < \bigtriangledown(x)$. Some aggressively optimizing compilers do not realize this distinction, and thus produce grossly incorrect code. The smallest floating point number x satisfying $0 < \bigtriangledown(x)$ is called the *machine eta*, and is denoted by η_M. This is the smallest positive subnormal number, and is thus equal to $\beta^{\check{e}-p+1} = \beta^{\check{e}} \varepsilon_M$. In Problem 2 of Computer Lab I the reader is asked to write a small program that computes both ε_M and η_M. These computations can safely be carried out in the default rounding mode, round to nearest even, since the least significant bits of both 1.0 and 0.0 is zero, which is an even number.

1.5 The IEEE standard

In the early days of computing, each brand of computer had its own implementation for floating point operations. This had the unfortunate effect that the outcome of a computation heavily depended on the precise type of machine used to perform the calculations. Naturally, this also severely limited the portability of programs, as code that worked perfectly well on one machine could crash on another. Even worse, most computer manufacturers had their own internal *representation* of floating point numbers. Of course, this led to the fact that data transfer between different machines became a highly complex task.

In the second half of the eighties, an international standard for the use and representation of floating point numbers was agreed upon. The standard is actually two standards: the IEEE p754, released in 1985, which deals exclusively with binary representations, and the IEEE p854, released in 1987, which in a common framework considers both base 2 and 10, see [IE85] and [IE87], respectively. Besides demanding maximal quality[7] of the arithmetic of floating point numbers, the standard also requires a consistent treatment of exceptions (e.g. division by zero) which greatly facilitates the construction of robust code. The IEEE standard also requires the presence of the four basic rounding modes: round up, round down, round to zero, and round to nearest (even).

For two very nice expositions of the IEEE floating point standard, see [Go91] and [Ov01].

1.5.1 The IEEE formats

The IEEE standard specifies two basic types of floating point numbers: the `single` and `double` formats. Extended versions of these formats are also mentioned, although only minimal requirements (as opposed to exact descriptions) for these are specified. Whereas the extended double format is provided on most architectures, the extended single format has found little support. We will therefore only cover the extended double format, which henceforth is denoted `extended`.

The standard also introduces three special symbols: `-Inf`, `+Inf`, and `NaN`. The two first are direct analogues to the mathematical notion of $\pm\infty$. Any finite floating point number x satisfies `-Inf` $< x <$`+Inf`. A real number greater that the largest finite floating point number is represented as either N^n_{max} or `+Inf`, depending on the rounding mode. Similarly, a real number smaller than the smallest finite floating point number is represented as either $-N^n_{max}$ or `-Inf`. The symbol `NaN`, which is short for *not a number*, is returned whenever the outcome of a floating point

[7] The IEEE standard requires that the basic operations $\{+, -, \times, \div\}$ and $\sqrt{}$ return the floating point nearest to the exact result, with regards to the rounding mode. Rather surprisingly, no such demands are imposed on the trigonometric and exponential functions. On a HP 9000/700, the argument $x = 2.50 \times 10^{17}$ produces the grossly incorrect $\sin x = 4.14415 \times 10^7$.

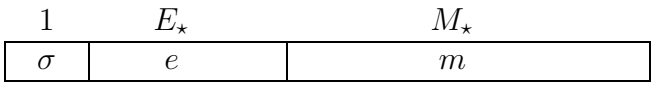


Figure 1.6: The basic IEEE formats.

operation is undefined, e.g. $0/0$ or `Inf` $-$ `Inf`.

To simplify the exposition, in all that follows, we will consider only floating point representations in binary base. Although this base has some special advantages, the theory presented can easily be generalized to a setting with an arbitrary base.

On almost all commercial computers, the two basic formats `single` and `double` are implemented using exactly the same mould, varying only two parameters. Each format is made up of a fixed number $F_\star$ of bits (binary integers), where the subscript $\star \in \{s, d\}$ indicates the specific format in question. The first bit encodes the sign of the floating point number, the following $E_\star$ bits correspond to the exponent, and the remaining $M_\star$ bits represent the mantissa. Thus, any two elements of $\{F_\star, E_\star, M_\star\}$ completely define the format in question. Actually, each format has precision $M_\star+1$. This is achieved by a *hidden bit*: since a normal floating point number represented in base two must start with a one, there is no need to explicitly store this leading bit. Note that this trick only works for binary representations. The exponent also has a little twist to it: we are not storing the actual exponent, but rather a *biased* version of it. Using $E_\star$ bits, we can represent any integer between 0 and $2^{E_\star}-1$. We form the actual exponent by subtracting the bias $B_\star = 2^{E_\star-1}-1$ from the stored number. This gives a exponent range of $[-2^{E_\star-1}+1, 2^{E_\star-1}]$. The two boundary points, however, are reserved for non-normal numbers.

Consider the $F_\star$-bit string $[\sigma; e_1e_2\ldots e_{E_\star}; m_1m_2\ldots m_{M_\star}]$, and let $E = (e_1e_2\ldots e_{E_\star})_2$ and $M = (0.m_1m_2\ldots m_{M_\star})_2$. Then the floating point number x represented by the string is decoded as follows:

(a) if $E = 2^{E_\star}-1$ and $M \neq 0$, then $x =$ `NaN`;

(b) if $E = 2^{E_\star}-1$ and $M = 0$, then $x = (-1)^\sigma$`Inf`;

(c) if $0 < E < 2^{E_\star}-1$, then $x = (-1)^\sigma 1.M \times 2^{E-B_\star}$;

(d) if $E = 0$ and $M \neq 0$, then $x = (-1)^\sigma 0.M \times 2^{1-B_\star}$;

(e) if $E = 0$ and $M = 0$, then $x = (-1)^\sigma \times 0$.

We see that case (c) corresponds to the set of normal numbers, whereas case (d) deals with the subnormal numbers. Note that cases (d) and (e) could be merged, although we chose not to do so seeing that zero is not a subnormal number.

The `single` format consists of 32 bits, of which eight bits correspond to the exponent, and the remaining 23 bits represent the mantissa. In other words, we have

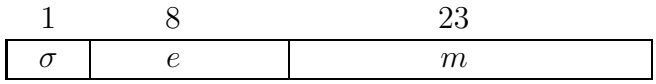


Figure 1.7: The IEEE `single` format.

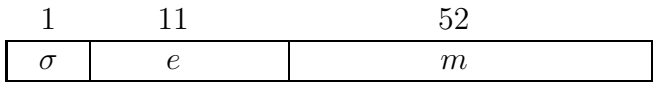


Figure 1.8: The IEEE `double` format.

$\{F_s, E_s, M_s\} = \{32, 8, 23\}$. The smallest and largest positive normal numbers in `single` format are seen to be $N^n_{min} = 2^{-126} \approx 1.2 \times 10^{-38}$ and $N^n_{max} = (2 - 2^{-23}) \times 2^{127} \approx 2^{128} \approx 3.4 \times 10^{38}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-23} \approx 1.2 \times 10^{-7}$. This means that we should expect about seven significant decimal digits.

The `double` format consists of 64 bits, of which 11 bits correspond to the exponent, and the remaining 52 bits represent the mantissa, i.e., we have $\{F_d, E_d, M_d\} = \{64, 11, 52\}$. The smallest and largest positive normal numbers in `double` format are seen to be $N^n_{min} = 2^{-1022} \approx 2.2 \times 10^{-308}$ and $N^n_{max} = (2 - 2^{-52}) \times 2^{1023} \approx 2^{1024} \approx 1.8 \times 10^{308}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-52} \approx 2.2 \times 10^{-16}$, so we can expect about 16 significant decimal digits from the `double` format.

1.5.2 The `extended` format

In contrast to the `single` and `double` formats, the IEEE standard does not specify absolute parameters for the `extended` format. Instead, minimal requirements are given, and it is up to each individual manufacturer to decide the precise parameters to be used. The requirements for the three formats are illustrated below.

Considering that the `extended` format is not precisely specified, it is rather unfortunate that it has become the most commonly used format. To make matters worse, the non-expert user is often unaware of this fact. The reason for this state of affairs

Table 1.1: The most common IEEE formats.

	`single`	`double`	`extended`
Format width in bits	32	64	≥ 79
Exponent width in bits	8	11	≥ 15
Precision p	24	53	≥ 64
Exponent bias	+127	+1023	unspecified
Maximal exponent	+127	+1023	$\geq +16383$
Minimal exponent	-126	-1022	≤ -16382

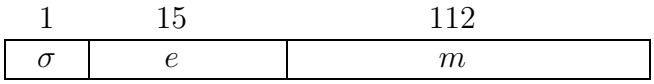


Figure 1.9: The SPARC 128-bit `extended` format.

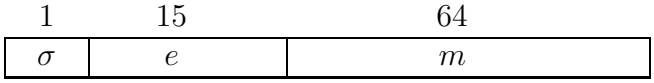


Figure 1.10: The Intel 80-bit `extended` format.

is that almost all computers perform intermediate computations in the widest registers available to them. Even the simplest computation, involving only two `double` type variables, will be converted to and performed in `extended` format, after which the result is rounded back to the `double` format. This can (and often does!) lead to quite unexpected results, which are very hard to "debug", see Example 1.6.1. In light of this, we will spend quite some time describing the various "flavours" of the `extended` format.

Let us begin with the 128-bit `extended` format, which is provided on the SPARC architecture. This format follows the generic mould described in the previous section, with parameters $\{F_e, E_e, M_e\} = \{128, 15, 112\}$. Thus, the smallest and largest positive normal numbers in the 128-bit `extended` format are seen to be $N^n_{min} = 2^{-16382} \approx 3.4\times10^{-4932}$ and $N^n_{max} = (2-2^{-112})\times2^{16383} \approx 2^{16384} \approx 1.2\times10^{4932}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-112} \approx 1.9 \times 10^{-34}$. This means that we should expect about 34 significant decimal digits from the 128-bit `extended` format.

In contrast to SPARC, the Intel x86 and Pentium architectures provide an 80-bit `extended` format, made up by ten 8-bit bytes[8]. This format consists of a 1-bit sign, a 15-bit (biased) exponent, and a 64-bit mantissa. In the 64-bit mantissa no hidden bit is employed, which leads to some minor peculiarities.

Consider the 80-bit string $[\sigma; e_1e_2\ldots e_{15}; m_0m_1m_2\ldots m_{63}]$. Let $E = (e_1e_2\ldots e_{15})_2$, and $M = (0.m_1m_2\ldots m_{63})_2$. Then the floating point number x represented by the string is decoded as follows:

(a) if $m_0 = 1$, $E = 32767$, and $M \neq 0$, then $x =$ `NaN`;

(b) if $m_0 = 1$, $E = 32767$, and $M = 0$, then $x = (-1)^\sigma$`Inf`;

(c) if $m_0 = 1$ and $0 < E < 32767$, then $x = (-1)^\sigma 1.M \times 2^{E-16383}$;

(d) if $m_0 = 0$, $E = 0$, and $M \neq 0$, then $x = (-1)^\sigma 0.M \times 2^{-16382}$;

(e) if $m_0 = 0$, $E = 0$, and $M = 0$, then $x = (-1)^\sigma \times 0$;

[8]Under the UNIX System V operating system, however, the format is made up by three 32-bit words, leaving the 16 highest addressed bits unused.

Table 1.2: Summary of the most common IEEE formats.

format	bits	m-bits	e-bits	N^s_{min}	N^n_{max}
`single`	32	23+1	8	1.4×10^{-45}	3.4×10^{38}
`double`	64	52+1	11	4.9×10^{-324}	1.8×10^{308}
INTEL `extended`	80	64	15	3.6×10^{-4951}	1.2×10^{4932}
SPARC `extended`	128	112+1	15	6.5×10^{-4966}	1.2×10^{4932}

(f) if $m_0 = 0$ and $0 < E < 32767$, then x is not defined;

(g) if $m_0 = 1$ and $E = 0$, then $x = (-1)^\sigma 1.M \times 2^{-16382}$;

The numbers corresponding to case (g) are called *pseudo-subnormal numbers*. These are never generated as results, but may appear as operands, in which case they are implicitly converted to the corresponding normal numbers as in (c).

The smallest and largest positive normal numbers in the 80-bit `extended` format are seen to be $N^n_{min} = 2^{-16382} \approx 3.4 \times 10^{-4932}$ and $N^n_{max} = (2 - 2^{-63}) \times 2^{16383} \approx 2^{16384} \approx 1.2 \times 10^{4932}$, respectively. The machine epsilon is $\varepsilon_M = 2^{-63} \approx 1.1 \times 10^{-19}$. This means that we should expect about 19 significant decimal digits from the 80-bit `extended` format.

The IBM S/390 G5 series, which uses a hexadecimal base, has hardware support for the IEEE formats with parameters matching those of the SPARC, see [SK99]. The CRAY T90 series has also moved toward the IEEE formats, but with some important exceptions. First, there is a slight linguistic discrepancy: the CRAY `single` format is 64 bits wide, and thus corresponds to the IEEE `double`. Similarly, the CRAY `double` format is 128 bits wide, and therefore corresponds to the IEEE `extended`. What is more serious is that the CRAY architecture does not treat subnormal numbers according to the IEEE standard. In fact, all subnormal numbers are forced to zero, see [Ga96]. As we have seen, this leads to several important mathematical laws being violated. To further confuse matters, the Macintosh PowerPC Numerics Environment provides a *double-double* format that is 128 bits wide. The exponent field, however, is only 11-bits wide, which is lower than the requirement for an IEEE `extended` format. The *double-double* is implemented in software combining two `double` formats in a quite complicated manner.

We end this section by listing some important facts about the various formats. Here, "m-bits" denotes the number of bits reserved for the mantissa, and "e-bits" corresponds to the exponent field.

1.6 Examples of floating point computations

A common way to determine the accuracy a specific computation is to gradually increase the precision until the result stabilizes. If adding more bits to the floating point representation does not alter the result of the computation, one usually accepts the result as being correct. The following example illustrates the false sense of security given by this approach.

Example 1.6.1 *Consider the function*

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y).$$

As observed in [Ru88], using FORTRAN on an IBM S/370 ($\beta = 16$), the function evaluated at the point $(\tilde{x}, \tilde{y}) = (77617, 33096)$ produces the following output:

type	p	$f(\tilde{x}, \tilde{y})$
`REAL*4`	*24*	$1.172603\ldots$
`REAL*8`	*53*	$1.1726039400531\ldots$
`REAL*10`	*64*	$1.172603940053178\ldots$

Using C or C++ (with gcc/g++) on an Intel Pentium III chip ($\beta = 2$), we get

type	p	$f(\tilde{x}, \tilde{y})$
`float`	*24*	1787028332140612812 80
`double`	*53*	1787028332140612812 80
`long double`	*64*	1787028332140612812 80

Using C or C++ (with gcc/g++) on a Sun UltraSPARC ($\beta = 2$), we get

type	p	$f(\tilde{x}, \tilde{y})$
`float`	*24*	25717841638407890822276893 9008
`double`	*53*	$1.17260394005317869492444060 59\ldots$
`long double`	*113*	$1.17260394005317869492444060 59\ldots$

Although all coefficients are exactly representable in base 2 (and thus in base 16), the rounding errors render the result useless. The correct answer is actually $-0.8273960599\ldots$, which means that we did not even get the sign right! These discrepancies are due to the fact that the two terms $T_1 = 5.5\tilde{y}^8$ and $T_2 = 333.75\tilde{y}^6 + \tilde{x}^2(11\tilde{x}^2\tilde{y}^2 - \tilde{y}^6 - 121\tilde{y}^4 - 2)$ are very large in modulus, and almost cancel:

$$T_1 = +7917111340668961361101134701524942848$$
$$T_2 = -7917111340668961361101134701524942850.$$

Since the sum of these terms is $T_1 + T_2 = -2$, we are left with just

$$f(\tilde{x}, \tilde{y}) = T_1 + T_2 + \tilde{x}/(2\tilde{y}) = -2 + \tilde{x}/(2\tilde{y}),$$

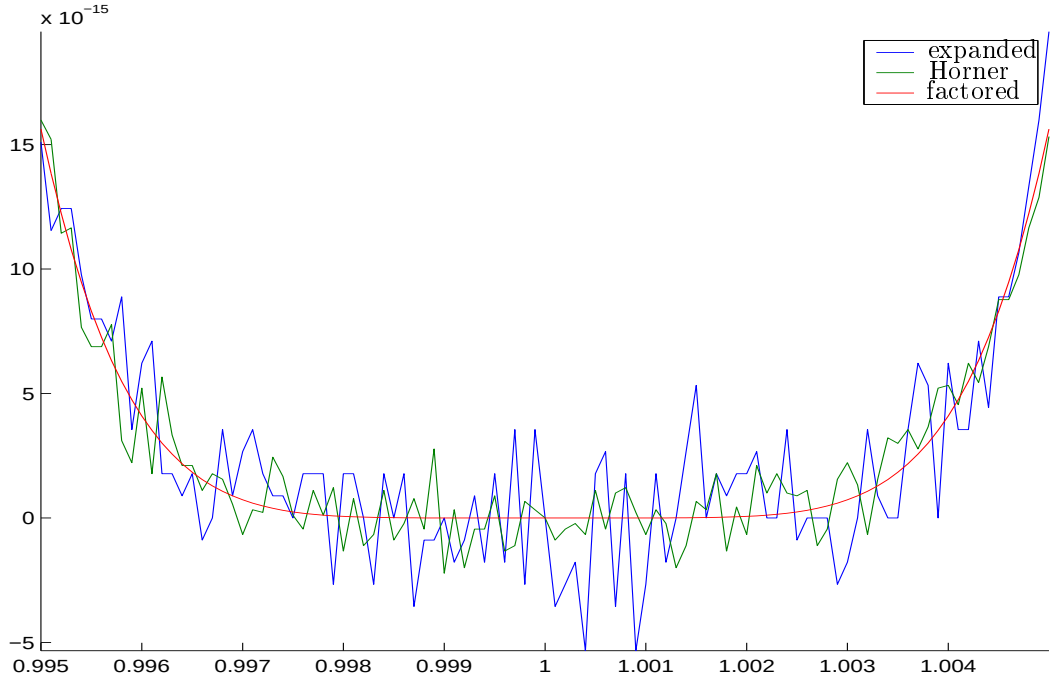


Figure 1.11: A smooth graph of a polynomial?

which gives

$$f(\tilde{x}, \tilde{y}) = -2 + \frac{77617}{2 \times 33096} \approx -0.8273960599.$$

Of course, when computing with any of the above-mentioned floating points formats, we have cancellation, and the sum of the two huge terms (both of magnitude $\approx 7.9 \times 10^{36}$*) is evaluated as zero. This results in the approximate function value*

$$f(\tilde{x}, \tilde{y}) = 0 + \frac{77617}{2 \times 33096} \approx 1.1726039400531.$$

This, however, does not explain the results from the Intel Pentium III chip. In this case, with no explicit instructions passed on to the compiler, all intermediate results are converted and worked upon in the `long double` *format by default.*

For a more thorough analysis of this example, see [EW02] and [CV01].

The next example deals with the (seemingly) simple task of generating the graph of a polynomial.

Example 1.6.2 *Plot the graph, and search for roots of the polynomial*

$$p(t) = t^6 - 6t^5 + 15t^4 - 20t^3 + 15t^2 - 6t + 1.$$

MATLAB produces the non-smooth graph illustrated in Figure 1.11. This picture is clearly wrong: a polynomial of degree n *can have at most* $n - 1$ *local extrema. Even*

if we minimize the number of floating point operations by evaluating the polynomial via Horner's method

$$p(t) = ((((((t-6)t+15)t-20)t+15)t-6)t+1)$$

the resulting graph is clearly not correct. Note, however, that we can factor the polynomial as $p(t) = (t-1)^6$*, which is (correctly) plotted as the smooth graph in Figure 1.11. It follows that the graph should lie above the* t*-axis, except at the multiple root* $t^* = 1$*.*

The reason why the computed values approximate the graph so poorly is that the condition number *of* $p(t)$ *near* $t^* = 1$ *is very large. Without going into explicit calculations, the condition number of* p *near* t^* *is given by*

$$\kappa(t) \approx 6 \left| \frac{t}{t - t^*} \right|.$$

As t *approaches the multiple root at* t^**, we see that* κ *tends to* $+\infty$*. A large condition numbers translates to poor accuracy, as is illustrated in Figure 1.11. Note that this situation would not occur if the multiple root was positioned at* 0 *rather than* 1*.*

The conclusion we draw is that function evaluations depend on the function *representation* when computed over $\mathbb{F}$. This is a difficult fact to accept for most mathematicians, who are used to computing over $\mathbb{R}$.

After these unnerving examples, let us show how one can obtain rigorous mathematical statements using the computer. By utilizing the directed rounding modes, we can enclose of the exact result of certain computations. These techniques will be generalized and studied in detail in Chapter 2.

Example 1.6.3 *It is well-known that the infinite series* $S = \sum_{k=1}^{\infty} k^{-2}$ *has the exact value* $\pi^2/6$*. Assume for the moment that we are unaware of this, and suppose that we need to find an approximation to* S*, say to 12 decimal places. Clearly, we cannot sum an infinite number of terms on the computer, so let us split the series into two pieces:*

$$S = \sum_{k=1}^{\infty} k^{-2} = \sum_{k=1}^{N} k^{-2} + \sum_{k=N+1}^{\infty} k^{-2} = S_N + S_N^\star.$$

Our strategy is now to bound the infinite part $S_N^\star$ *by mathematical means, whereas we calculate the finite part* S_N *using the computer.*

In order to achieve 12 *correct decimal places, we must ensure that the upper and lower bounds for* $S_N^\star$ *differ by at most* 5×10^{-13}*. By a simple geometric argument (draw the picture!), we have*

$$\int_{N+1}^{\infty} \frac{dx}{x^2} < S_N^\star < \int_{N+1}^{\infty} \frac{dx}{(x-1)^2},$$

which produces the bounds $\frac{1}{N+1} < S_N^\star < \frac{1}{N}$ *of width* $\delta_N = \frac{1}{N(N+1)}$. *Taking* $N = 2\times 10^6$ *gives* $\delta_N < 2.5 \times 10^{-13}$, *which should do nicely, assuming that we can compute the finite part* S_N *accurately.*

So let $N = 2 \times 10^6$, *and compute the sum* $S_N = \sum_{k=1}^{N} k^{-2}$ *with* `double` *precision, using the directed rounding modes. This produces the following output:*

```
Rounded down: S_N = 1.644 933 566 626 364 25
Rounded up  : S_N = 1.644 933 567 070 448 80.
```

As is plain to see, the results differ already in the 9th decimal. Since all terms are positive, however, the IEEE standard guarantees that the exact result is bounded from below by the result obtained when always rounding down. Analogously, the exact result is bounded from above by the result obtained when always rounding up. Thus we can enclose the exact value of S_N *in the interval*

$$[1.64493356662636425, 1.64493356707044880] \stackrel{\text{def}}{=} 1.64493356^{707044880}_{662636425}.$$

As the number of terms to be summed is known in advance, we can get better accuracy by adding the terms in increasing order (can you explain why?). Doing so yields the results

```
Rounded down: S_N = 1.644 933 566 848 350 46
Rounded up  : S_N = 1.644 933 566 848 352 46,
```

which now differ in the 15th decimal. Once again, we know that the exact value of S_N *is contained in the interval*

$$[1.64493356684835046, 1.64493356684835246] = 1.6449335668483^{246}_{046},$$

which allows us to refine our numerical enclosure of the partial sum:

$$S_N \in 1.6449335668483^{246}_{046} \subset 1.64493356^{707044880}_{662636425}.$$

Combining this information with the fact that $\frac{1}{N+1} < S_N^\star < \frac{1}{N}$ *produces the final enclosure:*

$$S \in 1.64493406684^{35253}_{10028},$$

which is correct to 12 decimal places. Compare this to the "exact" value $S = \pi^2/6 \approx 1.64493406684822630$, *where we have approximated* π *by its* 50 *leading digits.*

It is exactly this mixture of mathematics and properly rounded numerical computations that opens the door to *validated numerics*. As we have seen, these techniques allow us to construct computer-aided mathematical *proofs*, just like our recent proof that all digits of the approximation $S \approx 1.644934066848$ are correct.

1.7 Computer Lab I

Problem 1. Write a program that computes the factorial $n! \stackrel{\text{def}}{=} 1 \cdot 2 \cdot \cdots \cdot n$ of a given integer n. Use your program to compute the 40 first factorials. Do you notice anything strange? If so, try to explain what is happening.

Problem 2. Write a program that computes the smallest positive machine representable number η_M, and the machine epsilon ε_M. What are the values you get? Try to print them in hexadecimal form too (see the code segment below!).

Problem 3. (a) Find an IEEE double-precision floating point number $x \in (1, 2)$ such that $x \otimes \frac{1}{x} \neq 1$. (b) Find the smallest such number (possibly by a brute force search).

Problem 4. Define the function $f(x, y) = 9x^4 - y^4 + 2y^2$. Your objective is to compute $f(40545, 70226)$. Write a program that evaluates the function using each of the formats `int`, `float`, and `double`. What is the correct answer?

Problem 5. Write a program that switches the rounding mode on your computer. [Hint: for C/C++ programs, use the header file `round.h` listed in Section 2.4.2. MATLAB programs can use the file `setround.m` from Section 2.4.1.] Make your program compute 1/10 in various rounding modes. Make sure you output enough decimals, or even better – print the results in hexadecimal.

Problem 6. Write a small interval arithmetic routine supporting the arithmetic operations with directed rounding. Use the routine to compute $F([1, 2])$ and $G([1, 2])$, where $f(x) = \frac{7x-(x+1)^2}{3x}$ and $g(x) = \frac{7x-(x+1)^2}{3x-2}$, respectively.

```
/*   A function for printing in Hex format.   */
#include <iostream.h>
#include <stdio.h>

#ifdef __sparc     // Big Endian (SUN, IBM, Motorola).
#define HI_BITS 0
#define LO_BITS 1
#else              // Little Endian (Intel, Windows NT).
#define HI_BITS 1
#define LO_BITS 0
#endif

void printHex(const double &x) {
  printf("0x%08x 0x%08x",((int *) &x)[HI_BITS], ((int *) &x)[LO_BITS]);
}
```

Chapter 2

Interval arithmetic

In this chapter, we will briefly describe the fundamentals of interval arithmetic. We will also discuss how to implement the arithmetic in a programming environment.

Simply put, interval arithmetic is an arithmetic for *inequalities*. To illustrate this point, let us assume that we want to compute the area of a rectangle with side-lengths ℓ_1 and ℓ_2. Given the measurements $\ell_1 = 10.3 \pm 0.1$ and $\ell_2 = 4.4 \pm 0.2$, what can we say about the area $A = \ell_1 \cdot \ell_2$? If we express our measurements in terms of the bounds $|\ell_1 - 10.3| \leq 0.1$ and $|\ell_2 - 4.4| \leq 0.2$, then (using the triangle inequality) all we can say is that $|\ell_1 \cdot \ell_2 - 10.3 \cdot 4.4| \leq 0.2 \cdot 10.3 + 0.1 \cdot 4.4 + 0.1 \cdot 0.2$, i.e., $|A - 45.32| \leq 2.52$. If, on the other hand, we view the measure 2000 3-4 Numerical Methods for ODE 2004 ments as the inequalities $10.2 \leq \ell_1 \leq 10.4$ and $4.2 \leq \ell_2 \leq 4.6$, the optimal answer is obvious: the area must satisfy $42.84 = 10.2 \cdot 4.2 \leq A \leq 10.4 \cdot 4.6 = 47.84$, which translates into the slightly improved bound $|A - 45.34| \leq 2.5$.

The calculations in the latter case can be summarized as a single multiplication of two intervals:

$$[10.2, 10.4] \times [4.2, 4.6] = [42.84, 47.84].$$

Interval arithmetic justifies this extension of the real arithmetic, and provides an elegant means of computing with inequalities. For a concise reference on this topic, see e.g. [Mo66], [Mo79], or [AH83]. Early references are [Yo31], [Dw51], [Wa56],

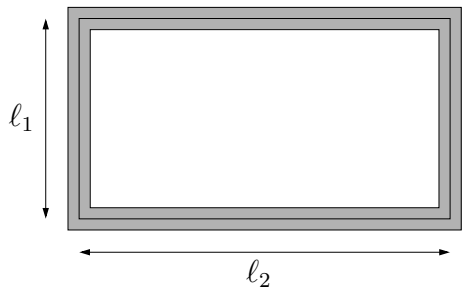


Figure 2.1: A rectangle with sides ℓ_1 and ℓ_2.

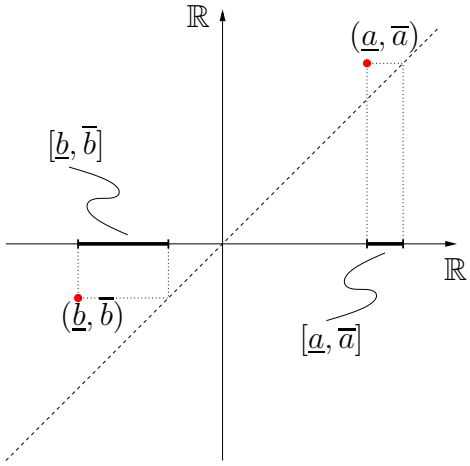


Figure 2.2: Identifying $\mathbb{IR}$ with $\{(x, y) \in \mathbb{R}^2 \colon y \geq x\}$.

[Su58] and [Mo59].

2.1 Real intervals

In what follows, our basic elements will be closed and bounded intervals of the real line. We will adopt the short-hand notation

$$[a] = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} \colon \underline{a} \leq x \leq \overline{a}\},$$

and consider the set of all such intervals of the real line:

$$\mathbb{IR} = \{[\underline{a}, \overline{a}] \colon \underline{a} \leq \overline{a}; \quad \underline{a}, \overline{a} \in \mathbb{R}\}.$$

Note that we allow for degenerate intervals $[a]$ with $\underline{a} = \overline{a}$. We will refer to these intervals as being *thin*. A natural embedding of $\mathbb{IR}$ in $\mathbb{R}^2$ is given by the mapping $g \colon \mathbb{IR} \to \mathbb{R}^2$, defined by $[\underline{a}, \overline{a}] \mapsto (\underline{a}, \overline{a})$. Geometrically, this corresponds to viewing $\mathbb{IR}$ as the region in $\mathbb{R}^2$ above and on the diagonal $y = x$. Points in $\mathbb{R}^2$ lying on the diagonal correspond to thin intervals.

Example 2.1.1 *The elements* $[-3, 4]$, $[1, 1]$, *and* $[\pi, 7]$ *all belong to* $\mathbb{IR}$, *whereas* $[2, -1]$ *and* $[-\infty, 0]$ *do not.*

Being sets, the elements of $\mathbb{IR}$ inherit the natural set relations, such as $=$, $\subseteq$, $\subset$, and $\overset{\circ}{\subset}$, defined by

$$\begin{aligned}
[a] = [b] \quad &\Leftrightarrow \quad \underline{a} = \underline{b} \text{ and } \overline{a} = \overline{b} \\
[a] \subseteq [b] \quad &\Leftrightarrow \quad \underline{b} \leq \underline{a} \text{ and } \overline{a} \leq \overline{b} \\
[a] \subset [b] \quad &\Leftrightarrow \quad [a] \subseteq [b] \text{ and } [a] \neq [b] \\
[a] \overset{\circ}{\subset} [b] \quad &\Leftrightarrow \quad \underline{b} < \underline{a} \text{ and } \overline{a} < \overline{b}
\end{aligned}$$

We can partially order[1] the set $\mathbb{IR}$ in several ways. Emphasizing the set-valued properties of $\mathbb{IR}$, we can use $\subseteq$ as a partial ordering. Preserving the the natural ordering of the real numbers, we may extend the relation $\leq$ to mean

$$[a] \leq [b] \quad \Leftrightarrow \quad \underline{a} \leq \underline{b} \text{ and } \overline{a} \leq \overline{b}.$$

This also provides a partial ordering of $\mathbb{IR}$.

By somewhat abusing our interval notation, we often identify a real number a with the corresponding thin interval $[a, a]$. It then makes sense to define the relation

$$a \in [b] \quad \Leftrightarrow \quad \underline{b} \leq a \text{ and } a \leq \overline{b},$$

which is really a special case of $[a] \subseteq [b]$. All of these relations can be complemented by their logical opposites $\neq$, $\not\subseteq$, $\not\subset$, $\overset{\circ}{\not\subset}$, and $\notin$.

We can also equip $\mathbb{IR}$ with analogues to the set operations $\cup$ and $\cap$. Both operations, however, require minor adjustments. First, taking the union of two intervals may not result in a new interval. To overcome this problem, we introduce the notion of forming the *hull* of two intervals:

$$[a] \sqcup [b] = [\min\{\underline{a}, \underline{b}\}, \max\{\overline{a}, \overline{b}\}].$$

It is clear that the resulting interval contains the union of $[a]$ and $[b]$. Second, the intersection of two intervals $[a]$ and $[b]$ is empty if either $\overline{a} < \underline{b}$ or $\overline{b} < \underline{a}$. Because of this, we must add the empty set (denoted by $[\emptyset]$) to $\mathbb{IR}$ for the intersection operator to be well-defined. When the intervals $[a]$ and $[b]$ have at least one point in common, the intersection is the standard one. Thus we have

$$[a] \cap [b] = \begin{cases} [\emptyset] & : \text{ if } \overline{a} < \underline{b} \text{ or } \overline{b} < \underline{a}, \\ [\max\{\underline{a}, \underline{b}\}, \min\{\overline{a}, \overline{b}\}] & : \text{ otherwise.} \end{cases}$$

Example 2.1.2 *Let* $[a] = [1, 3]$, $[b] = [1, \pi]$, $[c] = [-2.3, 4]$, *and* $[d] = [4, 5]$. *Then* $[a] \overset{\circ}{\subset} [c]$, $[a] \subset [b]$, $[a] \sqcup [b] = [1, \pi]$, $[a] \sqcup [d] = [1, 5]$, $[a] \cap [d] = [\emptyset]$, *and* $[c] \cap [d] = [4, 4]$.

Given an interval $[a] \in \mathbb{IR}$, we define the following real-valued functions

$$\begin{aligned} \text{rad}([a]) &= \tfrac{1}{2}(\overline{a} - \underline{a}) \qquad (\text{the } \textit{radius} \text{ of } [a]), \\ \text{mid}([a]) &= \tfrac{1}{2}(\overline{a} + \underline{a}) \qquad (\text{the } \textit{midpoint} \text{ of } [a]). \end{aligned}$$

Thus we can write $[a] = [\text{mid}([a]) - \text{rad}([a]), \text{mid}([a]) + \text{rad}([a])]$, and it follows that

$$\xi \in [x] \quad \Leftrightarrow \quad |\xi - \text{mid}([x])| \leq \text{rad}([x]),$$

[1] A relation $\sim$ is a *partial order* on a set S if, for all $a, b, c \in S$, it satisfies: (1) Reflexivity: $a \sim a$. (2) Antisymmetry: $a \sim b$ and $b \sim a$ implies $a = b$. (3) Transitivity: $a \sim b$ and $b \sim c$ implies $a \sim c$.

for any interval $[x]$. Two additional real-valued functions which often come in handy are

$$\begin{aligned} \operatorname{mig}([a]) &= \min\{|a| : a \in [a]\} \qquad (\text{the } \textit{mignitude} \text{ of } [a]), \\ \operatorname{mag}([a]) &= \max\{|a| : a \in [a]\} \qquad (\text{the } \textit{magnitude} \text{ of } [a]). \end{aligned}$$

These functions provide us with the smallest resp. largest distance to the origin attained by elements of $[a]$. There are explicit, computable formulas for these functions:

$$\operatorname{mig}([a]) = \begin{cases} 0 & : \text{ if } 0 \in [a], \\ \min\{|\underline{a}|, |\overline{a}|\} & : \text{ otherwise;} \end{cases} \qquad \operatorname{mag}([a]) = \max\{|\underline{a}|, |\overline{a}|\}.$$

Combining the two functions, we can form the *absolute value* of an interval:

$$\operatorname{abs}([a]) = \{|a| : a \in [a]\} = [\operatorname{mig}([a]), \operatorname{mag}([a])].$$

In contrast to the previously defined functions, the absolute value of an interval is an interval.

Example 2.1.3 *Let* $[x] = [-2, 3]$ *and* $[y] = [1, \pi]$. *Then* $\operatorname{mag}([x]) = 3$, $\operatorname{mig}([x]) = 0$, $\operatorname{mag}([y]) = \pi$, $\operatorname{mig}([y]) = 1$, $\operatorname{abs}([x]) = [0, 3]$, *and* $\operatorname{abs}([y]) = [1, \pi]$.

Finally, we can turn $\mathbb{IR}$ into a metric space[2] by equipping it with the Hausdorff distance:

$$d([a], [b]) = \max\{|\underline{a} - \underline{b}|, |\overline{a} - \overline{b}|\}. \tag{2.1}$$

Note that, according to our definitions, it follows that $d([a], [b]) = 0$ if and only if $[a] = [b]$. Using the metric, we can define the notion of a convergent sequence of intervals:

$$\begin{aligned} \lim_{k\to\infty} [a_k] = [a] \quad &\Leftrightarrow \quad \lim_{k\to\infty} d([a_k], [a]) = 0 \\ &\Leftrightarrow \quad \left(\lim_{k\to\infty} \underline{a}_k = \underline{a}\right) \wedge \left(\lim_{k\to\infty} \overline{a}_k = \overline{a}\right) \wedge \left(\forall k \quad \underline{a}_k \le \overline{a}_k\right). \end{aligned}$$

Note that the last condition is necessary for the $\Leftarrow$ direction.

2.2 Real interval arithmetic

In addition to viewing the elements of $\mathbb{IR}$ as sets, we may consider them as generalized real numbers. As such, it makes sense to attempt to define arithmetic on $\mathbb{IR}$. We have already seen that a copy of $\mathbb{R}$ is represented in $\mathbb{IR}$ as the set of thin intervals. It is therefore desirable to demand that the extended arithmetic should coincide with the normal real arithmetic for thin intervals. The most natural approach is to define binary arithmetic operations on elements of $\mathbb{IR}$ in a set theoretic manner:

[2] A metric space (X, d) is a pair consisting of a set X and a function $d\colon X \times X \to \mathbb{R}$ satisfying (1) $d(x, y) \ge 0$ for all $x, y \in X$. Also, $d(x, y) = 0$ if and only if $x = y$; (2) $d(x, y) = d(y, x)$ for all $x, y \in X$; and (3) $d(x, y) \le d(x, z) + d(z, y)$ for all $x, y, z \in X$.

Definition 2.2.1 *If $\star$ is one of the operators $+, -, \times, \div$, we define arithmetic on the elements of $\mathbb{IR}$ by*

$$[a] \star [b] = \{a \star b \colon a \in [a], b \in [b]\},$$

with the exception that $[a] \div [b]$ is undefined if $0 \in [b]$.

From this definition it is not immediately clear that the resulting set always is an interval. As we are working exclusively with closed intervals, however, it turns out that we can describe the resulting set in terms of the endpoints of the operands:

Proposition 2.2.2 *Arithmetic on the elements of $\mathbb{IR}$ is given by*

$$\begin{aligned}
[a] + [b] &= [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \\
[a] - [b] &= [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \\
[a] \times [b] &= [\min\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}, \max\{\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}\}] \\
[a] \div [b] &= [a] \times [1/\overline{b}, 1/\underline{b}], \quad \textit{if } 0 \notin [b].
\end{aligned}$$

Proof: The reason why the resulting set is an interval is due to the fact that any *real* operation $+, -, \times, \div$ is continuous in both of its arguments, with the exception of dividing by zero (this is why $[a] \div [b]$ is undefined[3] if $0 \in [b]$). If we fix one of the arguments, the real operations are monotone in the remaining argument. The monotonicity implies that extremal values are attained on the boundary of the domains, i.e., at the endpoints of the intervals. The proposition can thus be verified by examining a finite number of cases. □

As a consequence of Proposition 2.2.2, it follows that $\mathbb{IR}$ is an arithmetically closed subset of $\mathcal{P}(\mathbb{R})$ – the power set[4] of the real numbers.

From a computer programming point of view, this is good news indeed: using the formulas from Proposition 2.2.2, it is straight-forward to implement the datatype `interval` with its associated arithmetic, see Section 2.4. From a practical perspective, the formulas for multiplication and division can be made more efficient. As it stands, a single *interval* multiplication requires four *real* multiplications (as well as several comparisons). This number can be reduced by checking the sign of each endpoint of the two intervals. It is easy to see that interval multiplication can be divided into nine cases, as illustrated in Figure 2.3. Only one case requires four real multiplications; the other cases require just two.

As an example, assume that $0 \leq \underline{a} \leq \overline{a}$ and $\underline{b} \leq 0 \leq \overline{b}$. This situation corresponds to the square on the second row, third column in Figure 2.3. It is clear that the maximal element of $[a] \times [b] = \{a \times b \colon a \in [a], b \in [b]\}$ is given by choosing the largest elements from both $[a]$ and $[b]$. By the same token, the minimal element of $[a] \times [b]$

[3] We can allow for division by zero by extending the underlying set of real numbers to include the concept of infinity. We will address this topic in Section 2.3.

[4] The power set $\mathcal{P}(\mathbb{S})$ of a set $\mathbb{S}$ is the set of all subsets of $\mathbb{S}$.

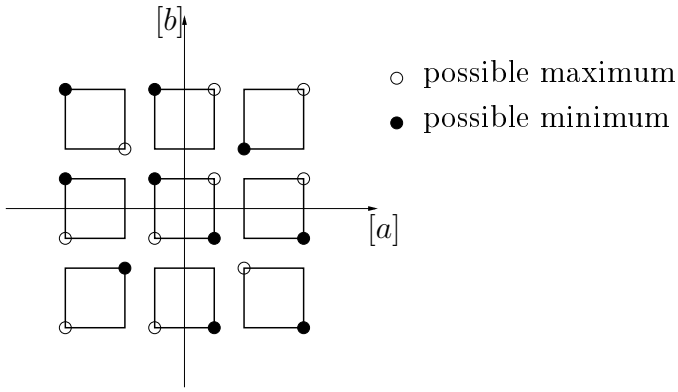


Figure 2.3: A more efficient interval multiplication scheme.

is given by choosing the largest element from $[a]$ and the smallest element from $[b]$. The resulting interval is thus given by $[a] \times [b] = [\overline{a}\underline{b}, \overline{a}\overline{b}]$, which only requires two real multiplications. In a similar fashion, the formula for interval division can be reduced to six simpler cases.

Example 2.2.3 *Using Proposition 2.2.2, we can compute*

$$\begin{aligned} [-1,0] + [0,\pi] &= [-1,\pi] & \qquad [-1,-1] \times [2,5] &= [-5,-2] \\ [1,4] - [1,4] &= [-3,3] & \qquad [-2,3] \times [-2,3] &= [-6,9] \\ [\tfrac{1}{2},1] - [0,\tfrac{1}{6}] &= [\tfrac{1}{3},1] & \qquad [1,\sqrt{2}] \times [-1,1] &= [-\sqrt{2},\sqrt{2}] \\ [2,4] - [3,3] &= [-1,1] & \qquad [1,2] \div [-2,-1] &= [-2,-\tfrac{1}{2}]. \end{aligned}$$

It follows from Definition 2.2.1 (or from Proposition 2.2.2) that addition and multiplication are both associative and commutative: for $[a], [b], [c] \in \mathbb{IR}$, we have

$$\begin{aligned} [a] + ([b] + [c]) &= ([a] + [b]) + [c]; & \qquad [a] + [b] &= [b] + [a], \\ [a] \times ([b] \times [c]) &= ([a] \times [b]) \times [c]; & \qquad [a] \times [b] &= [b] \times [a]. \end{aligned}$$

Also, it is clear that the elements $[0,0]$ and $[1,1]$ are the unique neutral elements with respect to addition and multiplication, respectively. Note, however, that in general an element in $\mathbb{IR}$ has no additive or multiplicative inverse. For example, we have $[1,2] - [1,2] = [-1,1] \neq [0,0]$, and $[1,2] \div [1,2] = [\frac{1}{2},2] \neq [1,1]$. As a consequence, the distributive law does *not* always hold. As an example[5], we have

$$[-1,1]([-1,0] + [3,4]) = [-1,1][2,4] = [-4,4],$$

whereas

$$[-1,1][-1,0] + [-1,1][3,4] = [-1,1] + [-4,4] = [-5,5].$$

[5]Here, and in what follows, we will often suppress the multiplication symbol $\times$.

This unusual property is important to keep in mind when representing functions as part of an interval calculation. Interval arithmetic satisfies a weaker rule than the distributive law, which we shall refer to as *sub-distributivity*:

$$[a]([b]+[c]) \subseteq [a][b]+[a][c]. \tag{2.2}$$

This is a set theoretical property that illustrates one of the fundamental differences between real- and interval arithmetic.

Exercise 2.2.4 *Prove that the space* $\mathbb{IR}$ *can be partially ordered by either relation* $\subseteq$ *or* $\leq$*, as described in Section 2.1.*

Exercise 2.2.5 *Prove that* $[a]([b]+[c]) = [a][b]+[a][c]$ *when either*

(1) $[a]$ *is thin.*

(2) all elements of $[b]$ *and* $[c]$ *have the same sign.*

Exercise 2.2.6 *Given an interval* $[a]$ *show that*

(1) $0 \in [a]-[a]$*, but that in general* $[a]-[a] \neq [0,0]$*,*

(2) $1 \in [a] \div [a]$*, but that in general* $[a] \div [a] \neq [1,1]$*. (Assume* $0 \notin [a]$*.)*

Another key feature of interval arithmetic is that of *inclusion isotonicity*:

Theorem 2.2.7 *If* $[a] \subseteq [a']$*,* $[b] \subseteq [b']$*, and* $\star \in \{+,-,\times,\div\}$*, then*

$$[a] \star [b] \subseteq [a'] \star [b'],$$

where we demand that $0 \notin [b']$ *for division.*

This is the single most important property of interval arithmetic: it allows us to accurately estimate the range of a large class of functions. This will be explained in a later section. Note that, in particular, Theorem 2.2.7 holds when $[a]$ and $[b]$ are thin intervals, i.e., real numbers.

Proof: It is somewhat amazing that this powerful theorem has a classical "one-line" proof: by an immediate application of Definition 2.2.1, we have

$$[a] \star [b] = \{a \star b\colon a \in [a], b \in [b]\} \subseteq \{a \star b\colon a \in [a'], b \in [b']\} = [a'] \star [b'].$$

□

2.3 Extended interval arithmetic

According to Definition 2.2.1, we cannot divide by an interval containing zero. Nevertheless, if we attempt to reinterpret the spirit of the formula

$$[a] \div [b] = \{a \div b \colon a \in [a], b \in [b]\},$$

as

$$[a] \div [b] = \{c \in \mathbb{R} \colon bc = a,\ a \in [a], b \in [b]\}, \tag{2.3}$$

there might be a way around this slight imperfection. The procedure, however, is quite delicate, and implementing it on a computer raises some subtle questions regarding how we choose to extend the real numbers to include the concept of infinity. Before going into details, let us illustrate the use of (2.3) in a simple setting.

Example 2.3.1 *If $a = [1, 2]$ and $b = [-5, 3]$, then according to (2.3), the quotient $[c] = [a] \div [b]$ is given by*

$$[c] = \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in [-5, 3]\}.$$

Focusing on the particular value $b = 0$, we want to find all c such that $0 \cdot c \in [1, 2]$. As the equation clearly has no solution, we loose no information by discarding this case. Hence

$$\begin{aligned}
[c] &= \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in [-5, 0) \cup (0, 3]\} \\
&= \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in [-5, 0)\} \cup \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in (0, 3]\} \\
&= \big([1, 2] \div [-5, 0)\big) \cup \big([1, 2] \div (0, 3]\big).
\end{aligned}$$

The first set may be interpreted as the limit

$$\begin{aligned}
[c]^- &= \lim_{\varepsilon \to 0^-} \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in [-5, \varepsilon)\} \\
&= \lim_{\varepsilon \to 0^-} [1, 2] \div [-5, \varepsilon) = \lim_{\varepsilon \to 0^-} (\tfrac{2}{\varepsilon}, -\tfrac{1}{5}) = (-\infty, -\tfrac{1}{5}].
\end{aligned}$$

Similarly, the second set may be interpreted as the limit

$$\begin{aligned}
[c]^+ &= \lim_{\varepsilon \to 0^+} \{c \in \mathbb{R} \colon bc = a,\ a \in [1, 2], b \in (\varepsilon, 3]\} \\
&= \lim_{\varepsilon \to 0^+} [1, 2] \div (\varepsilon, 3] = \lim_{\varepsilon \to 0^+} [\tfrac{1}{3}, \tfrac{2}{\varepsilon}) = [\tfrac{1}{3}, \infty).
\end{aligned}$$

Combining the two results, we have the answer

$$[1, 2] \div [-5, 3] = (-\infty, -\tfrac{1}{5}] \cup [\tfrac{1}{3}, \infty) = \mathbb{R} \setminus (-\tfrac{1}{5}, \tfrac{1}{3}).$$

This example indicates that we need a notion of infinity in order to perform the extended interval division. There are several ways we can allow for "division by zero" – it all boils down to how we choose to extend the real numbers. From a mathematical point of view, there are more or less elegant extensions. We will acquaint ourselves with three variants, which are appropriately named: *the good*, *the bad*, and *the ugly*. Naturally, we shall stick to the ugly.

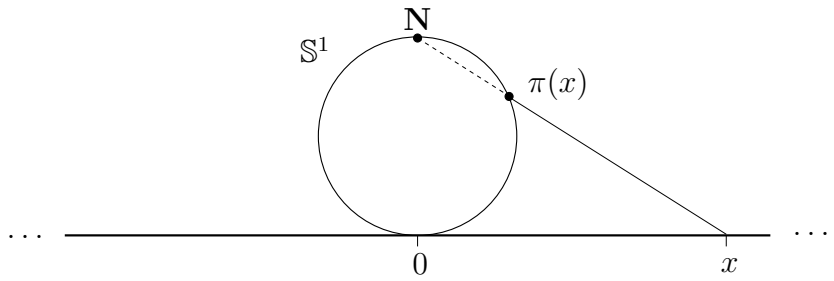


Figure 2.4: Identifying $\mathbb{R}^*$ and $\mathbb{S}^1$ via the projection $\pi\colon \mathbb{R}^* \to \mathbb{S}^1$.

2.3.1 The good: projective extension

The projective extension of the real numbers, usually denoted $\mathbb{R}^*$, is formed by adding the unsigned "point at infinity" ∞ to the real line. This one-point compactification of the real line allows us to identify $\mathbb{R}^*$ with the closed circle $\mathbb{S}^1$ where the north-pole $\mathbf{N}$ plays the role of infinity, see Figure 2.4.

We can partially extend the arithmetic operations from $\mathbb{R}$ to $\mathbb{R}^*$ in the following manner:

$$\begin{aligned} -(\infty) &= \infty, & x + \infty = \infty + x &= \infty \text{ if } x \neq \infty, \\ x \cdot \infty = \infty \cdot x &= \infty \text{ if } x \neq 0, & x/\infty &= 0 \text{ if } x \neq \infty, \\ x/0 &= \infty \text{ if } x \neq 0. \end{aligned}$$

The expressions $\infty \pm \infty$, ∞/∞ and $0 \cdot \infty$, however, are undefined.

In this setting there is no need for the interpretation (2.3). Instead, repeating the division performed in Example 2.3.1, we immediately have

$$\begin{aligned} [1,2] \div [-5,3] &= [1,2] \div \big([-5,0) \cup \{0\} \cup (0,3]\big) \\ &= \big([1,2] \div [-5,0)\big) \cup \big([1,2] \div 0\big) \cup \big([1,2] \div (0,3]\big) \\ &= \{x \in \mathbb{R}\colon x \leq -\tfrac{1}{5}\} \cup \{\infty\} \cup \{x \in \mathbb{R}\colon \tfrac{1}{3} \leq x\}. \end{aligned}$$

Note that we no longer can write $(-\infty, -\frac{1}{5})$ for $\{x \in \mathbb{R}\colon x \leq -\frac{1}{5}\}$. This is because, in the projective extension, we have $-\infty = \infty$, i.e., there is only one infinity, and it cannot be compared to the finite real numbers with any of the relations $\{<, \leq, >, \geq\}$. As a consequence, we cannot assign the value zero to an expression like $e^{-\infty}$, since in $\mathbb{R}^*$ the equality $e^{-\infty} = e^{\infty}$ must hold. On the other hand, it makes perfect sense to write $\tan(\pi/2) = \infty$. The beautiful part of the projective extension is the way we can represent the result of a "division by zero". So far, the outcome of performing $[1,2] \div [-5,3]$ appears to be a rather messy expression. Nevertheless, using the topology of the circle, we may adopt a short-hand notation for *extended intervals*:

$$[\tfrac{1}{3}, -\tfrac{1}{5}] = \{x \in \mathbb{R}\colon x \leq -\tfrac{1}{5}\} \cup \{\infty\} \cup \{x \in \mathbb{R}\colon \tfrac{1}{3} \leq x\}.$$

To motivate this notation, we refer to Figure 2.5, which shows two intervals in $\mathbb{R}^*$. The interval containing zero is simply the set $\{x \in \mathbb{R}^*\colon -3 \leq x \text{ and } x \leq 2\}$, which

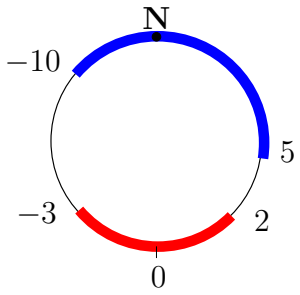


Figure 2.5: The two intervals $[-3, 2]$ and $[5, -10]$ in $\mathbb{R}^*$.

we denote $[-3, 2]$ as usual. The second interval, however, is different. It represents the set $\{x \in \mathbb{R}^* : x \leq -10 \text{ or } 5 \leq x \text{ or } x = \infty\}$, which we can write as $[5, -10]$. This should be interpreted on the circle as moving from the left endpoint, 5, counter-clockwise to the right endpoint, -10, just as with normal intervals. Unfortunately, there is no suitable way to represent the extended real line $\mathbb{R}^*$ in this manner.

Motivated by the preceeding discussion, we define the set $\mathbb{IR}^*$ of projectively extended intervals:

$$\mathbb{IR}^* = \{[\underline{a}, \overline{a}] : \quad \underline{a}, \overline{a} \in \mathbb{R}^*\},$$

where the case $\underline{a} > \overline{a}$ is interpreted as an extended interval.

2.3.2 The bad: affine extension

The affine extension of the real numbers, usually denoted $\overline{\mathbb{R}}$, is formed by adding the two signed infinities, $-\infty$ and $+\infty$, to the real line. This two-point compactification of the real line allows us to write $\overline{\mathbb{R}}$ as the closed interval $[-\infty, +\infty]$. The arithmetic operations can be partially extended to $\overline{\mathbb{R}}$ in the following manner:

$$\begin{aligned} -(+\infty) = -\infty \text{ and } -(-\infty) = +\infty, \qquad & x + (+\infty) = +\infty \text{ if } x \neq -\infty, \\ x + (-\infty) = -\infty \text{ if } x \neq +\infty, \qquad & x \cdot (\pm\infty) = \pm\infty \text{ if } x > 0, \\ x \cdot (\pm\infty) = \mp\infty \text{ if } x < 0, \qquad & x/(\pm\infty) = 0 \text{ if } x \neq \pm\infty. \end{aligned}$$

The expressions $+\infty + (-\infty)$, $-\infty + (+\infty)$, and $x/0$, however, are undefined. In contrast to the projective extension, the affine infinities can be compared in size: $-\infty < x < +\infty$ if $x \neq \pm\infty$ and $-\infty < +\infty$. Furthermore, the affine extension has the appealing property that it makes perfect sense to write statements like $e^{-\infty} = 0$, $e^{+\infty} = +\infty$, $\ln 0 = -\infty$, and $\ln(+\infty) = +\infty$. This property makes the affine extension the preferred choice among analysists.

It is now straight-forward to define the set $\overline{\mathbb{IR}}$ of affinely extended intervals:

$$\overline{\mathbb{IR}} = \{[\underline{a}, \overline{a}] : \underline{a} \leq \overline{a}; \quad \underline{a}, \overline{a} \in \overline{\mathbb{R}}\}.$$

Thus, apart from the elements of $\mathbb{IR}$, also intervals on the form $[-\infty, x]$, $[x, +\infty]$, and $[-\infty, +\infty]$ are valid elements of $\overline{\mathbb{IR}}$.

Since we cannot divide by zero in $\overline{\mathbb{R}}$, repeating the division performed in Example 2.3.1 requires the interpretation (2.3), and produces

$$[1,2] \div [-5,3] = [-\infty, -\tfrac{1}{5}] \cup [\tfrac{1}{3}, \infty].$$

As with the projective extension, we could simply introduce the notion of extended intervals (thus removing the demand $\underline{a} \leq \overline{a}$ for intervals), now with the meaning

$$[\tfrac{1}{3}, -\tfrac{1}{5}] = [-\infty, -\tfrac{1}{5}] \cup [\tfrac{1}{3}, \infty]. \tag{2.4}$$

Alternatively, we could accept the fact that some interval operations may produce a union of intervals. This line of action, however, leads to some tricky implementation issues. A yet simpler way of resolving the whole issue would be to return the entire line $[-\infty, +\infty]$ when dividing by zero, possibly with the exception that $[a]/[0] = [\emptyset]$ if $0 \notin [a]$. As this approach leads to an unneccessary loss of information, it is therefore not very widespead. We will use the definition

$$\overline{\mathbb{IR}} = \{[\underline{a}, \overline{a}] \colon \underline{a}, \overline{a} \in \overline{\mathbb{R}}\}$$

where the case $\underline{a} \geq \overline{a}$ corresponds to an extended interval of type (2.4).

2.3.3 The ugly: signed zero

While all elements of $\mathbb{R}^*$ have unique reciprocals, this is not the case for all members of $\overline{\mathbb{R}}$. Indeed, in $\overline{\mathbb{R}}$ we have $1/(-\infty) = 1/(+\infty) = 0$, whereas $1/0$ is undefined. In an attempt to resolve this problem, it is possible to equip $\overline{\mathbb{R}}$ with *signed zeroes*, satisfying $x/(+0) = \operatorname{sign}(x) \cdot (+\infty)$ and $x/(-0) = \operatorname{sign}(x) \cdot (-\infty)$ for $x \neq \pm 0$. In effect, this gives both infinites and both signed zeroes unique reciprocals, just like all other elements of $\overline{\mathbb{R}}$:

$$1/(+\infty) = +0, \qquad 1/(+0) = +\infty, \qquad 1/(-\infty) = -0, \qquad 1/(-0) = -\infty.$$

As an illustration, we have $1 \div [+0, 2] = [\frac{1}{2}, +\infty]$, whereas $1 \div [-0, 2] = [-\infty, +\infty]$. Unfortunately, there is no natural way of propagating the sign of the zero under addition and subtraction: what sign should $(+0) + (-0)$ or even $x - x$ have?

The IEEE standard incorporates signed infinities as well as signed zeroes. The signs appear naturally within the actual sign-exponent-mantissa encoding of the floating point numbers. Even though -0 and $+0$ are distinct values, they both compare as equal, and are only distinguishable by comparing their sign bits. Regarding addition and subtraction, the standard [IE85] states

> When the sum of two operands with opposite signs (or the difference of two operands with equal signs) is exactly zero, the sign of that sum (or difference) shall be + in all rounding modes except round toward $-\infty$, in which mode that sign shall be $-$. However, $x + x = x - (-x)$ retains the same sign as x even when x is zero.

Thus, on any computer compliant with the IEEE standard, we have $(+0)+(-0) = x - x = +0$, unless we are rounding with $\bigtriangledown$, answering the question posed above.

The signed zeroes were not introduced for their mathematical elegance: their presence is due to computer manufacturers' desire to reduce the number of fatal floating point errors. Instead of having to abort a computation that happens to perform a division by zero, it is much more desirable to produce a well-defined result of the division. Without the signed zero, this is simply not possible.

2.3.4 The extended interval division

In light of the previous discussion, the extended interval division is defined over the space $\overline{\mathbb{IR}}$ (equipped with signed zeros), where we allow for extended intervals of the form (2.4). Following [Ra96], we define division over $\overline{\mathbb{IR}}$ as follows

$$[a] \div [b] = \begin{cases} [a] \times [1/\overline{b}, 1/\underline{b}] & \text{if } 0 \notin [b], \\ [-\infty, +\infty] & \text{if } 0 \in [a] \text{ and } 0 \in [b], \\ [\overline{a}/\underline{b}, +\infty] & \text{if } \overline{a} < 0 \text{ and } \underline{b} < \overline{b} = 0, \\ [\overline{a}/\underline{b}, \overline{a}/\overline{b}] & \text{if } \overline{a} < 0 \text{ and } \underline{b} < 0 < \overline{b}, \\ [-\infty, \overline{a}/\overline{b}] & \text{if } \overline{a} < 0 \text{ and } 0 = \underline{b} < \overline{b}, \\ [-\infty, \underline{a}/\underline{b}] & \text{if } 0 < \underline{a} \text{ and } \underline{b} < \overline{b} = 0, \\ [\underline{a}/\overline{b}, \underline{a}/\underline{b}] & \text{if } 0 < \underline{a} \text{ and } \underline{b} < 0 < \overline{b}, \\ [\underline{a}/\overline{b}, +\infty] & \text{if } 0 < \underline{a} \text{ and } 0 = \underline{b} < \overline{b}, \\ [\emptyset] & \text{if } 0 \notin [a] \text{ and } [b] = [0, 0]. \end{cases} \tag{2.5}$$

Case 1 deals with non-zero divisors, although it now incorporates quotients such as $[6, 8] \div [2, +\infty] = [+0, 4]$. Cases 4 and 7 yield extended intervals, i.e., these particular results actually consist of a union of two infinite intervals. In [Ra96], it is proved that the division defined by (2.5) is inclusion isotonic, i.e., if $[a] \subseteq [a']$, and $[b] \subseteq [b']$, then $[a] \div [b] \subseteq [a'] \div [b']$, which generalizes Theorem 2.2.7.

It is worth pointing out that, although signed zeros are not explicitly present in (2.5), their properities are mimicked in the formulas. As an example, let us consider case 5, where the condition is stated as "if $\overline{a} < 0$ and $0 = \underline{b} < \overline{b}$". For all practical purposes, this can be interpreted as "if $\overline{a} < 0$ and $+0 = \underline{b} < \overline{b}$".

There are two main advantages of having access to an extended interval division in a computing environment. First, all run-time errors of the type *division by zero* are immediately avoided. Usually, an error of this type will cause a program to crash, unless some serious error-handling capabilities have been provided by the programmer. Second, it is actually desirable, from a mathematical point of view, to be able to perform extended division. Later on, we will see a striking example of this when we study the interval Newton method.

2.3.5 Containment sets

In many situations, not even an extended interval arithmetic will suffice. In a set-valued environment there are many more ways in which a seemingly innocent calculation can result in a run-time error. As an example, consider the function $f(x) = \sqrt{x - x^2}$ evaluated over the domain $[x] = [0, 1]$. It is clear that the exact range[6] of f is well-defined on this domain: $R(f; [0, 1]) = [0, 1/\sqrt{2}]$. The interval evaluation, however, is problematic:

$$F([0,1]) = \sqrt{[0,1] - [0,1]^2} = \sqrt{[0,1] - [0,1]} = \sqrt{[-1,1]}.$$

There are (at least) three ways to handle this situation: (1) we can abort the computation; (2) we can give a complex-valued result; (3) we can restrict the domain before taking the square root. The idea of containment sets (csets) is to use something similar to alternative (3), that is, for any domain $[x]$, we define

$$\sqrt{[x]} = \sqrt{[x] \cap [0, +\infty]}.$$

For the example considered above, this yields:

$$F([0,1]) = \sqrt{[-1,1]} = \sqrt{[-1,1] \cap [0,+\infty]} = \sqrt{[0,1]} = [0,1].$$

The idea is to disregard arguments that do not belong to the function's natural domain. Note that, although we do not obtain the exact range of the function, we still have an enclosure: $R(f; [0, 1]) \subseteq F([0, 1])$. This is paramount to interval compuations, and will be further addressed in Section 3.1.

Let us now approach the strategy more systematically. The aim is to create a completely *exception free* system, i.e., one where all operations are always well-defined. Given an elementary, real-valued function $f\colon D_f \to \mathbb{R}$, where D_f is the largest domain on which f is well-defined, we introduce the cset extension $f^*\colon \mathcal{P}\overline{\mathbb{R}} \to \mathcal{P}\overline{\mathbb{R}}$ via

$$f^*(S) = R(f; S \cap D_f) \cup \{\lim_{\zeta \to \zeta^*} f(\zeta) \colon \zeta \in D_f, \zeta^* \in S \setminus D_f\}. \tag{2.6}$$

Here $\overline{\mathbb{R}}$ is the set of affinely extended reals $\overline{\mathbb{R}} = \{-\infty\} \cup \mathbb{R} \cup \{+\infty\}$, and $\mathcal{P}\overline{\mathbb{R}}$ is the set of all subsets of $\overline{\mathbb{R}}$ – the power set of $\overline{\mathbb{R}}$. The definition (2.6) says that we consider the cset extension to be made up of two parts: one part for all points that belong to f's natural domain D_f, and one part for points that are located just outside D_f.

Returning to the example above, we have $f(x) = g(x - x^2)$, where $g(x) = \sqrt{x}$ with $D_g = [0, +\infty)$, and $S = [-1, 1]$. Using (2.6), we get

$$\begin{aligned} g^*([-1,1]) &= R(g; [-1,1] \cap [0,+\infty)) \cup \{\lim_{\zeta \to \zeta^*} g(\zeta) \colon \zeta \in [0,+\infty), \zeta^* \in [-1,1] \setminus [0,+\infty)\} \\ &= R(g; [0,1]) \cup \{\lim_{\zeta \to \zeta^*} g(\zeta) \colon \zeta \in [0,+\infty), \zeta^* \in [-1,0)\} \\ &= g([0,1]) \cup \emptyset = \sqrt{[0,1]} = [0,1], \end{aligned}$$

[6] The range is defined as $R(f; S) = \{f(x) \colon x \in S\}$.

which corresponds to a pure domain restriction.

A more interesting example is given by taking $h(x) = 1/x$, which has the natural domain $D_h = (-\infty, 0) \cup (0, +\infty)$. Again, by using (2.6), we get for $S = \{0\}$

$$\begin{aligned} h^*(0) &= R(h; \{0\} \cap \{(-\infty, 0) \cup (0, +\infty)\}) \\ &\quad \cup \{\lim_{\zeta \to \zeta^*} h(\zeta) \colon \zeta \in (-\infty, 0) \cup (0, +\infty), \zeta^* \in \{0\} \setminus \{(-\infty, 0) \cup (0, +\infty)\}\} \\ &= R(h; \emptyset) \cup \{\lim_{\zeta \to 0} h(\zeta) \colon \zeta \in (-\infty, 0) \cup (0, +\infty)\} \\ &= \emptyset \cup \{\lim_{\zeta \to 0^-} h(\zeta)\} \cup \{\lim_{\zeta \to 0^+} h(\zeta)\} = \{-\infty\} \cup \{+\infty\}. \end{aligned}$$

Compare this result to the last line of (2.5) which defines the outcome as the empty set. We can redo Example 2.3.1 by taking $S = [-3, 5]$:

$$\begin{aligned} h^*([-3, 5]) &= R(h; [-3, 5] \cap \{(-\infty, 0) \cup (0, +\infty)\}) \\ &\quad \cup \{\lim_{\zeta \to \zeta^*} f(\zeta) \colon \zeta \in (-\infty, 0) \cup (0, +\infty), \zeta^* \in [-3, 5] \setminus \{(-\infty, 0) \cup (0, +\infty)\}\} \\ &= R(h; [-3, 0) \cup (0, 5]) \cup \{\lim_{\zeta \to 0} h(\zeta) \colon \zeta \in (-\infty, 0) \cup (0, +\infty)\} \\ &= (-\infty, -\tfrac{1}{3}] \cup [\tfrac{1}{5}, +\infty) \cup \{\lim_{\zeta \to 0^-} h(\zeta)\} \cup \{\lim_{\zeta \to 0^+} h(\zeta)\} \\ &= (-\infty, -\tfrac{1}{3}] \cup [\tfrac{1}{5}, +\infty) \cup \{-\infty\} \cup \{+\infty\} = [-\infty, -\tfrac{1}{3}] \cup [\tfrac{1}{5}, +\infty]. \end{aligned}$$

This is exactly the same result that was obtained in Section 2.3.2.

Now, for an interval version of csets, we can return the interval hull of the result. Using the examples above, we get $F^*([0, 1]) = G^*([-1, 1]) = [0, 1]$, $H^*([0, 0]) = H^*([-3, 5]) = [-\infty, +\infty]$. A really nice feature of this definition is that the interval cset arithmetic is exception free: that is, there are no undefined operations. As a consequence, we always get a true enclosure of the range:

$$R(f; [x]) \subseteq F^*([x]).$$

For a clear exposition of theoretical as well as practical points of containment sets, see [PC06]. SUN Microsystems' `C++` and Fortran compilers support interval arithmetic with cset functionality, see [1]. The free `C++` library `FILIB++` also has this functionality as an option, see [LT06].

Exercise 2.3.2 *A special case of Brouwer's fixed point theorem ([Br10]) states that any continuous function $f \colon [-1, 1] \to [-1, 1]$ has a fixed point x^* in $[-1, 1]$, i.e., $f(x^*) = x^*$. If $f(x) = \sqrt{x} - 1$, and F^* is the cset extension of f, then we have $R(f; [-1, 1]) \subseteq F^*([-1, 1]) = [-1, 0]$ (show this!). This means that f indeed maps $[-1, 1]$ into itself, and by Brouwer's theorem f should have a fixed point in $[-1, 1]$, which it does not. Explain!*

2.4 Floating point interval arithmetic

When implementing interval arithmetic on a computer, we no longer work over the space $\mathbb{R}$, but rather $\mathbb{F}$ - the floating point numbers of the computer. This is a finite set, and so is $\mathbb{IF}$ – the set of all intervals whose endpoints belong to $\mathbb{F}$:

$$\mathbb{IF} = \{[\underline{a}, \overline{a}] \colon \underline{a} \leq \overline{a}; \quad \underline{a}, \overline{a} \in \mathbb{F}\}.$$

As discussed earlier, $\mathbb{F}$ is not arithmetically closed. Thus, when performing arithmetic on intervals in $\mathbb{F}$ we must round the resulting interval *outwards* to guarantee inclusion of the true result. By this, we mean that the lower bound is rounded down, and the upper bound is rounded up. For $[a], [b] \in \mathbb{IF}$, we define

$$\begin{aligned}
[a] + [b] &= [\bigtriangledown(\underline{a} + \underline{b}), \bigtriangleup(\overline{a} + \overline{b})] \\
[a] - [b] &= [\bigtriangledown(\underline{a} - \overline{b}), \bigtriangleup(\overline{a} - \underline{b})] \\
[a] \times [b] &= [\min\{\bigtriangledown(\underline{a}\underline{b}), \bigtriangledown(\underline{a}\overline{b}), \bigtriangledown(\overline{a}\underline{b}), \bigtriangledown(\overline{a}\overline{b})\}, \\
&\qquad \max\{\bigtriangleup(\underline{a}\underline{b}), \bigtriangleup(\underline{a}\overline{b}), \bigtriangleup(\overline{a}\underline{b}), \bigtriangleup(\overline{a}\overline{b})\}] \\
[a] \div [b] &= [\min\{\bigtriangledown(\underline{a}/\underline{b}), \bigtriangledown(\underline{a}/\overline{b}), \bigtriangledown(\overline{a}/\underline{b}), \bigtriangledown(\overline{a}/\overline{b})\}, \\
&\qquad \max\{\bigtriangleup(\underline{a}/\underline{b}), \bigtriangleup(\underline{a}/\overline{b}), \bigtriangleup(\overline{a}/\underline{b}), \bigtriangleup(\overline{a}/\overline{b})\}], \quad \text{if } 0 \notin [b].
\end{aligned}$$

Recall that $\bigtriangledown(x)$ and $\bigtriangleup(x)$ were defined in Section 1.3.2. The resulting type of arithmetic is called interval arithmetic with *directed rounding*. As we shall see, this is easily implemented on a computer that supports the directed roundings.

With regards to efficiency, a single $\mathbb{IF}$-multiplication requires eight $\mathbb{F}$-multiplications: four products must be computed under two different rounding modes. As before, it is customary to break the formula for multiplication into nine cases (depending of the signs of the operands' endpoints). Out of these nine cases, only one will involve four $\mathbb{F}$-multiplications; the remaining eight will need just two. In a similar manner, the (non-extended) $\mathbb{IF}$-division can be split into six cases.

Exercise 2.4.1 *Derive the optimal formulas for division in $\mathbb{IF}$, assuming that a floating point comparison is much faster than a floating point division.*

Extending the floating point interval arithmetic via (2.5) is straight-forward, and yields the set

$$\overline{\mathbb{IF}} = \{[\underline{a}, \overline{a}] \colon \underline{a}, \overline{a} \in \overline{\mathbb{F}}\},$$

where the case $\underline{a} \geq \overline{a}$ corresponds to an extended interval of type (2.4).

2.4.1 A `MATLAB` implementation of interval arithmetic

To illustrate how easy it is to get started, we present a simple `MATLAB` implementation of (non-extended) interval arithmetic with directed rounding.

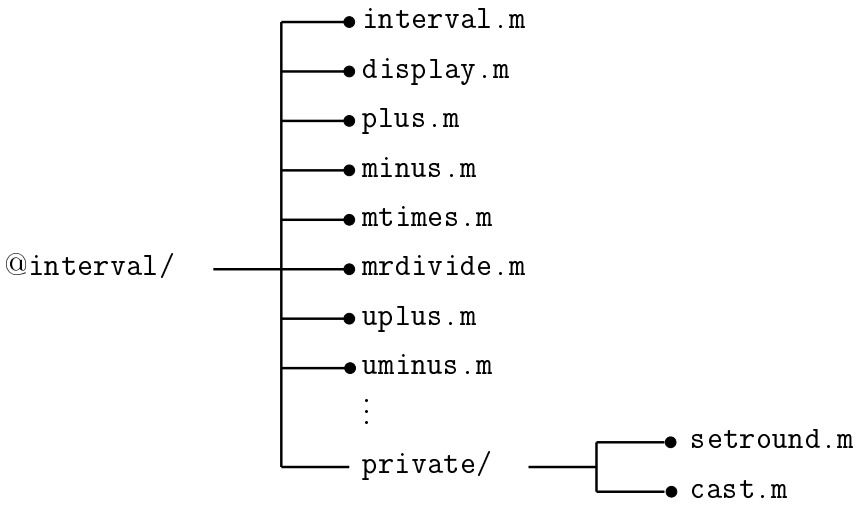


Figure 2.6: The hierarchy of the MATLAB `interval` class.

As most modern programming languages, MATLAB uses *classes* to define new data types, and *methods* to define the functionality of a user-defined class. A new class can be added to the MATLAB environment by specifying a structure that provides data storage for the object, and creating a class directory containing m-files[7] that operate on the object. These m-files contain the methods for the class. MATLAB is somewhat peculiar in that it demands a certain file hierarchy associated with each class. In Figure 2.6 we illustrate a simple setup for our `interval` class. We will explain the purpose of the different m-files as we go along.

As we want to build an `interval` class, we begin by creating a directory called `@interval` where all m-files associated to the `interval` class will reside. Having done this, we create the m-file `interval.m`, in which we define what is meant by an interval. It is natural to implement an interval as a class consisting of two numbers – the endpoints of the interval:

```
01 function iv = interval(lo, hi)
02 % A naive interval class constructor.
03 if nargin == 1
04     hi = lo;
05 elseif ( hi < lo )
06     error('The endpoints do not define an interval.');
07 end
08 iv.lo = lo; iv.hi = hi;
09 iv = class(iv,'interval');
```

By including lines 03 and 04, we allow the constuctor to automatically convert a single number x to a thin interval $[x, x]$. Note that, as opposed to most programming languages, MATLAB only supports the `double` format, which means that no explicit type declarations have to (or can!) be made.

[7] An m-file is simply a text file `filename.m` containing a sequence of MATLAB statements to be executed. The file extension of `.m` makes this a MATLAB m-file.

Operation	m-file	New description
`a + b`	`plus(a,b)`	Interval addition
`a - b`	`minus(a,b)`	Interval subtraction
`a * b`	`mtimes(a,b)`	Interval multiplication
`a / b`	`rmdiv(a,b)`	Interval division
`+a`	`uplus(a)`	Unary plus
`-a`	`uminus(a)`	Unary minus

Table 2.1: Overloaded `MATLAB` arithmetic methods.

We must also inform `MATLAB` how to display `interval` objects. This is achieved via the m-file `display.m`:

```
01 function display(iv)
02 % A simple output formatter for the interval class.
03 disp([inputname(1), ' = ']);
04 fprintf('  [%17.17f, %17.17f]\n', iv.lo, iv.hi);
```

We can now input/output intervals within the `MATLAB` environment:

```
>> a = interval(3, 4), b = interval(2, 5), c = interval(1)
a =
  [3.00000000000000000, 4.00000000000000000]
b =
  [2.00000000000000000, 5.00000000000000000]
c =
  [1.00000000000000000, 1.00000000000000000]
```

When creating user-defined classes, it is often desirable to change the behavior of the `MATLAB` operators and functions[8] for cases when the arguments are user-defined classes. This can be accomplished by *overloading* the relevant functions. Overloading enables a function to handle different types and numbers of input arguments, and perform whatever operation is appropriate for the situation at hand.

Each native `MATLAB` operator has an associated function name (e.g., the + operator has an associated `plus.m` function). Any such operator can be overloaded by creating an m-file with the appropriate name in the class directory. In Table 2.4.1, we list the operators we intend to overload in our `interval` class.

Here, another feature of `MATLAB` becomes evident: the `MATLAB`-engine regards all numeric elements as matrices, even if they are single numbers. Indeed, a single number can be viewed as a 1×1-matrix.

Let us begin by writing a function that returns the sum of two intervals:

```
01 function result = plus(a, b)
```

[8] In what follows, we will not distinguish between *functions* and *methods*.

```
02 % Overloading the '+' operator for intervals.
03 [a, b] = cast(a, b);
04 setround(-inf);
05 lo = a.lo + b.lo;
06 setround(+inf);
07 hi = a.hi + b.hi;
08 setround(0.5);
09 result = interval(lo, hi);
```

Let us examine this small piece of code: First, the function `cast`, appearing on line 03, makes sure that the inputs `a` and `b` are intervals. If one of them is not an interval, it is converted to an interval by a call the interval constructor, see the listing below.

```
01 function [a, b] = cast(a, b)
02 % Casts non-intervals to intervals.
03 if ~isa(a, 'interval')
04     a = interval(a);
05 end
06 if ~isa(b, 'interval')
07     b = interval(b);
08 end
```

Casting[9] allows for expressions like $[1,2]+3$, which is converted to $[1,2]+[3,3]$, and evaluated to $[4,5]$.

Second, the function `setround`, appearing on lines 04, 06, and 08 of the file `plus.m`, instructs the `MATLAB`-engine to switch the rounding direction before performing an arithmetic operation. This function is implemented[10] in the auxiliary file `setround.m`, presented below:

```
01 function setround(rnd)
02 % A switch for changing rounding mode. The arguments
03 % {+inf, -inf, 0.5, 0} correspond to the roundings
04 % {upward, downward, to nearest, to zero}, respectively.
05 system_dependent('setround',rnd);
```

We consider both functions `cast` and `setround` to be intrinsic to the `interval` class. By placing their m-files in the `private` subdirectory, these functions are hidden from non-interval classes.

Carrying on, it is straight-forward to write functions that overload the remaining arithmetic operations $-$, $\times$, and $\div$. Below, we present a `MATLAB` listing of the division algorithm:

```
01 function result = mrdivide(a, b)
02 % A non-optimal interval division algorithm.
03 [a, b] = cast(a, b);
```

[9] In the programming language `C++`, casting is implicitly performed at the compilation stage. This simplifies the actual programming, but can also produce hard-to-find bugs.

[10] Unfortunately, this feature is not available on SPARC platforms. Instead, a small `C` program must pre-compiled into a so called `mex`-file, see the code in Section 2.4.2.

```
04 if ( (b.lo <= 0.0) & (0.0 <= b.hi) )
05     error('Denominator straddles zero.');
06 else
07     setround(-inf);
08     tmp1 = min(a.lo / b.lo, a.lo / b.hi);
09     tmp2 = min(a.hi / b.lo, a.hi / b.hi);
10     lo = min(tmp1, tmp2);
11     setround(+inf);
12     tmp1 = max(a.lo / b.lo, a.lo / b.hi);
13     tmp2 = max(a.hi / b.lo, a.hi / b.hi);
14     hi = max(tmp1, tmp2);
15     setround(0.5);
16     result = interval(lo, hi);
17 end
```

Performing some simple interval calculations, we have:

```
>> a+b, a-b, a*b, a/b
ans =
  [5.00000000000000000, 9.00000000000000000]
ans =
  [-2.00000000000000000, 2.00000000000000000]
ans =
  [6.00000000000000000, 20.00000000000000000]
ans =
  [0.59999999999999998, 2.00000000000000000]
```

The outward rounding is apparent in the left endpoint of the last result. All other endpoints were computed exactly. We should point out that our interval constructor is still very rudimentary, and does not handle user-input adequately. As an example, suppose we would like to generate the smallest interval containing 1/10. As a first attempt, we may try something like

```
>> interval(1/10)
ans =
  [0.10000000000000001, 0.10000000000000001]
```

which is *not* what we wanted. The problem here is that the quotient 1/10 is *first* rounded to a single floating point number, which is *then* converted to a thin interval. Since 1/10 has no exact representation in the floating point format, we obtain an interval that does not contain 1/10. A way to work around this is to declare either the nominator or denominator as an interval. Since integers have exact representations, no rounding takes place at this stage. It is only when the division takes place that the directed rounding kicks in, producing a non-thin interval straddling 1/10:

```
>> interval(1)/10
ans =
  [0.09999999999999999, 0.10000000000000001]
```

More sophisticated interval libraries provide a means for entering strings of numbers[11], such as

[11] The `MATLAB` package `IntLab` has this functionality, as does the `C++` toolbox `CXSC`, see [INv4] and [CXSC], respectively.

```
>> interval('1/10')
ans =
  [0.09999999999999999, 0.10000000000000001]
```

Nevertheless, this has a cost in programming effort which we are not willing to pay at the moment.

Continuing our calculations, we can now illustrate the sub-distributive property of interval arithmetic:

```
>> c = interval(0.25, 0.50)
c =
  [0.25000000000000000, 0.50000000000000000]
>> a*(b+c), a*b+a*c
ans =
  [5.25000000000000000, 22.00000000000000000]
ans =
  [5.00000000000000000, 22.00000000000000000]
```

Notice the differing lower endpoints; clearly the expression $ab+ac$ produces a wider result than $a(b+c)$. Since all computations in this example are exact, the outward rounding does not affect the result.

Exercise 2.4.2 *Modify the appropriate m-files so they perform multiplication and division by checking the signs of the operands' endpoints.*

Exercise 2.4.3 *Add some interval functions (e.g.* `sin(x)` *and* `pow(x,n)`*) to the* `interval` *class. Note that this requires some knowledge of how accurate the corresponding real-valued functions are in the underlying programming environment.*

Exercise 2.4.4 *Do you know any other programming language that supports operator overloading? If so, try to implement a rudimentary interval arithmetic library whose syntax permits expressions like* `x + y` *and* `z = sin(1 + pow(x,2))`*, where* `x`*,* `y`*, and* `z` *are of type* `interval`*.*

Now that we have all arithmetic operations in place, let us consider the built-in relational operators provided by `MATLAB`. Some of these are listed in Table 2.4.1.

When overloading these methods, we will give them new, interval-based, meanings. Let us begin with the simplest of them all: the *equality* relation. Two intervals are *equal* exactly when their endpoints agree. Analogously, two intervals are *not equal* if at least one of their endpoints differ. Both functions can be implemented in a few lines.

```
01 function result = eq(a, b)
02 % The '(e)qual' operator '=='.
03 [a, b] = cast(a, b);
04 result = ( (a.lo == b.lo) & (a.hi == b.hi) );
```

Operation	m-file	New description
`a == b`	`eq(a,b)`	Equal to
`a ~= b`	`ne(a,b)`	Not equal to
`a <= b`	`le(a,b)`	Subset of
`a < b`	`lt(a,b)`	Proper subset of
`a & b`	`and(a,b)`	Intersection
`a | b`	`or(a,b)`	Interval hull

Table 2.2: Overloaded `MATLAB` set-relation methods.

```
01 function result = ne(a, b)
02 % The '(n)ot (e)qual' operator '~='.
03 [a, b] = cast(a, b);
04 result = ( (a.lo ~= b.lo) | (a.hi ~= b.hi) );
```

Turning to the order-relations *less or equal* and *less than*, we will interpret them as the set-relations *inclusion* $\subseteq$ and *proper inclusion* $\mathring{\subset}$, respectively.

```
01 function result = le(a, b)
02 % The '(l)ess or (e)qual' operator '<='. Means 'a inside b'.
03 [a, b] = cast(a, b);
04 result = ( (b.lo <= a.lo) & ( a.hi <= b.hi) );
```

```
01 function result = lt(a, b)
02 % The '(l)ess (t)han' operator '<'. Means 'a inside int(b)',
03 [a, b] = cast(a, b);
04 result = ( (b.lo < a.lo) & ( a.hi < b.hi) );
```

The four functions we have defined so far are all *boolean*, i.e., their return-values come from the set $\{\texttt{true}, \texttt{false}\}$. In `MATLAB` (and most other programming languages), these alternatives are coded as '1' and '0', respectively.

```
>> a = interval(1, 10); b = interval(-2, 3); c = interval(3, 5);
>> [a==b, a==c, a~=b, a~=c, a<=b, b<=a, a<c, c<a]
ans =
     0     0     1     1     0     0     0     1
```

Finally, we will implement the logical operatiors *and* and *or*, but we will re-define them as the set-operations *intersection* $\cap$, and *hull* $\sqcup$, respectively. One complication here is that two intervals may have an empty intersection. Seeing that our simple interval constructor does not accommodate empty intervals, we will return the `MATLAB` version of the empty set, accompanied by a warning[12] whenever this situation occurs.

[12]It may be desirable to comment out the warning in order to minimize unnecessary output.

```
01 function result = and(a, b)
02 % The 'and' operator '&'. Means 'a intersected with b'.
03 [a, b] = cast(a, b);
04 if ( (a.hi < b.lo) | (b.hi < a.lo) )
05     warning('The intervals do not intersect.');
06     result = [];
07 else
08     result = interval(max(a.lo, b.lo), min(a.hi, b.hi));
09 end
```

Since we have not defined the interval methods to operate on empty sets, it is vital that we have a means for detecting an empty set. Fortunately, `MATLAB` has a built-in function `isempty` that can reveal whether the outcome of an interval-intersection is an empty set or not.

```
>> a=interval(1,3); b=interval(4,5); c=interval(2,5);
>> aANDb = a & b; aANDc = a & c;
Warning: The intervals do not intersect.
> In /matlab/@interval/and.m at line 5
aANDb =
     []
aANDc =
  [2.00000000000000000, 3.00000000000000000]
>> [isempty(aANDb), isempty(aANDc)]
ans =
     1     0
```

The hull of two intervals is always well-defined, and thus straight-forward to implement.

```
01 function result = or(a, b)
02 % The 'or' operator '|'. Means 'hull of a and b'.
03 [a, b] = cast(a, b);
04 result = interval(min(a.lo, b.lo), max(a.hi, b.hi));
```

```
>> aORb = a | b, aORc = a | c
aORb =
  [1.00000000000000000, 5.00000000000000000]
aORc =
  [1.00000000000000000, 5.00000000000000000]
```

Exercise 2.4.5 *Make the necessary modifications to the m-files* `interval.m` *and* `display.m` *to accommodate input/output of empty intervals. As a first step you must find a good representation for the empty interval.*

Exercise 2.4.6 *How would you modify the remaining m-files to fully incorporate empty intervals?*

Exercise 2.4.7 *Implement a new class* `xinterval` *for extended intervals. Try to extend all associated interval methods described in this section. [Warning: this takes some effort!]*

Exercise 2.4.8 *An interval can also be represented by the two numbers* $< m, r >$, *where* m *is the midpoint, and* r *is the radius of the interval. Derive the interval-arithmetic rules using only these two quantities. From a numerical point of view, what are the merits/drawbacks of this (*`midrad`*) representation compared to the end-point (*`infsup`*) representation?*

```
/*   File: round.h

     A header file that defines the directed
     rounding modes for Linux and Sparc.
*/

#if defined(__linux)
#include <fenv.h>
#define ROUND_DOWN  FE_DOWNWARD
#define ROUND_UP    FE_UPWARD
#define ROUND_NEAR  FE_TONEAREST
#define setRound    fesetround
#endif

#if defined(__sparc)
#include <ieeefp.h>
#define ROUND_DOWN  FP_RM
#define ROUND_UP    FP_RP
#define ROUND_NEAR  FP_RN
#define setRound    fpsetround
#endif

void setRoundDown() { setRound(ROUND_DOWN); }
void setRoundUp  () { setRound(ROUND_UP);   }
void setRoundNear() { setRound(ROUND_NEAR); }
```

Figure 2.7: A header for switching rounding mode within `C/C++` codes.

2.4.2 Changing rounding modes

Here we present a header file that allows the rounding direction to be changed from within a `C/C++` program.

As mentioned in Section 2.4.1 all platforms do not support the `MATLAB` command `system_dependent`. In order to switch rounding mode one can instead compile a small `C` program containing the switching instructions into a module that `MATLAB` can use. This is done via `MATLAB`'s compiler `mex`.

```
/*   File: setround.c

     A simple C program that allows directed rounding
     from within the matlab environment.

     Compilation: mex setround.c
*/
#include "mex.h"
#include "round.h"

void round(double *In)
{
    if ( In[0] == mxGetInf() )
        setRoundUp();
    else if ( In[0] == -mxGetInf() )
        setRoundDown();
    else if ( In[0] == 0.0 )
        setRoundZero();
    else if ( In[0] == 0.5 )
        setRoundNear();
    if ( In[0] != 0 && In[0] != 0.5 && !mxIsInf(In[0]) && !mxIsInf(-In[0]) )
        mexErrMsgTxt("Only +inf, -inf, 0, 0.5 are valid arguments");
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    mxArray *In_ptr;
    double  *In;

    In_ptr = (mxArray *)prhs[0];
    In = mxGetPr(In_ptr);

    if ( nrhs != 1 )
        mexErrMsgTxt("Pick an argument from the set {+inf, -inf, 0, 0.5}.");
    else if ( nlhs != 0 )
        mexErrMsgTxt("No output argument allowed!");
    round(In);
}
```

Figure 2.8: A mex file for switching rounding mode within `MATLAB` codes.

Chapter 3

Interval analysis

3.1 Interval functions

One of the main points of studying interval arithmetic is that we want a simple way of enclosing the *range* of a real-valued function.

Definition 3.1.1 *Let $D \subseteq \mathbb{R}$, and consider a function $f\colon D \to \mathbb{R}$. We define the range of f over D to be the set*

$$R(f; D) = \{f(x)\colon x \in D\}.$$

Except for the most trivial cases, mathematics provides few tools to describe the range of a given function f over a specific domain D. Indeed, today there exists an entire branch of mathematics and computer science – Optimization Theory – devoted to "simply" finding the smallest element of the set $R(f; D)$. We shall see that interval arithmetic provides a helping hand in this matter.

As a first step, we begin by attempting to extend the real functions to *interval functions*. By this, we mean functions who take and return intervals rather than real numbers. We already have the theory to extend rational functions, i.e., functions on the form $f(x) = p(x)/q(x)$, where p and q are polynomials. Simply substituting all occurrences of the real variable x with the interval variable $[x]$ produces a rational interval function $F([x])$, called the *natural* interval extension of f.

Theorem 3.1.2 *Given a real-valued, rational function f, and its natural interval-extension F such that $F([x])$ is well-defined for some $[x] \in \mathbb{IR}$, we have*

(1) $[z] \subseteq [z'] \subseteq [x] \Rightarrow F([z]) \subseteq F([z'])$, *(inclusion isotonicity)*

(2) $R(f; [x]) \subseteq F([x])$. *(range enclosure)*

Proof: We demand that $F([x])$ be well-defined simply to avoid domain violations as duscussed in Section 2.3.5. Part (1) then follows by repeatedly invoking Theorem 2.2.7, while evaluating F. To see that part (2) follows, we note that f and F are related by the equality $F([x,x]) = f(x)$. Now, assume that $R(f;[x]) \not\subseteq F([x])$. Then there is a $\zeta \in [x]$ such that $f(\zeta) \in R(f;[x])$ but $f(\zeta) \notin F([x])$. This, however, implies that $f(\zeta) = F([\zeta,\zeta]) \notin F([x])$, which violates part (1). Hence $R(f;[x]) \subseteq F([x])$. □

We now want to treat more general functions than the rational ones. Monotone functions are easily extended: it suffices to evaluate the endpoints of the interval argument. To illustrate this, we may define

$$\begin{array}{rcllll} e^{[x]} & = & \exp[x] & = & [e^{\underline{x}}, e^{\overline{x}}] & \\ \sqrt{[x]} & = & \mathrm{sqrt}[x] & = & [\sqrt{\underline{x}}, \sqrt{\overline{x}}] & \text{if } 0 \le \underline{x} \\ & & \log[x] & = & [\log \underline{x}, \log \overline{x}] & \text{if } 0 < \underline{x} \\ & & \arctan[x] & = & [\arctan \underline{x}, \arctan \overline{x}]\ . & \end{array}$$

Simple, non-monotone functions are also easily handled if they only have a finite number of extremæ, which are known. As an example, we have

$$[x]^n = \mathrm{pow}([x],n) = \begin{cases} [\underline{x}^n, \overline{x}^n] & : \text{ if } n \in \mathbb{Z}^+ \text{ is odd,} \\ [\mathrm{mig}([x])^n, \mathrm{mag}([x])^n] & : \text{ if } n \in \mathbb{Z}^+ \text{ is even,} \\ [1,1] & : \text{ if } n = 0, \\ [1/\overline{x}, 1/\underline{x}]^{-n} & : \text{ if } n \in \mathbb{Z}^- \text{ and } 0 \notin [x]. \end{cases}$$

Note that this definition provides tighter enclosures than the naive approach $[x] \times [x] \times \cdots \times [x]$ (with n factors). This is because basic interval arithmetic does not distinguish between variables and their domains. To illustrate this important point, let us consider the square, $f(x) = x^2$. If $[x] = [-2,3]$, then $[x]\times[x] = [-6,9]$, whereas $R(x^2;[-2,3]) = [0,9]$. Our definition of the power of an interval, however, recognizes that there is only one variable (and not n variables with coinciding domains) and thus produces a *sharp* enclosure of the range, i.e., $R(x^n;[a]) = [a]^n$. Naturally, we always have $[a]^n \subseteq [a] \times [a] \times \cdots \times [a]$.

Trigonometric functions are handled in a similar manner. As an example, if we define $S^+ = \{2k\pi + \pi/2 \colon k \in \mathbb{Z}\}$ and $S^- = \{2k\pi - \pi/2 \colon k \in \mathbb{Z}\}$, we have

$$\sin[x] = \begin{cases} [-1,1] & : \text{ if } [x] \cap S^- \neq [\emptyset] \text{ and } [x] \cap S^+ \neq [\emptyset], \\ [-1, \max\{\sin \underline{x}, \sin \overline{x}\}] & : \text{ if } [x] \cap S^- \neq [\emptyset] \text{ and } [x] \cap S^+ = [\emptyset], \\ [\min\{\sin \underline{x}, \sin \overline{x}\}, 1] & : \text{ if } [x] \cap S^- = [\emptyset] \text{ and } [x] \cap S^+ \neq [\emptyset], \\ [\min\{\sin \underline{x}, \sin \overline{x}\}, \max\{\sin \underline{x}, \sin \overline{x}\}] & : \text{ if } [x] \cap S^- = [\emptyset] \text{ and } [x] \cap S^+ = [\emptyset]. \end{cases}$$

Computing the interval version of $\sin x$ is thus reduced to determining whether the sets S^+ and S^- intersect the compact domain $[x]$ or not, which is, in principle[1], not

[1] An important impementation issue is that if the interval $[x]$ is located very far away from the origin, then the knowledge of a large number of leading digits of π is required.

hard. Note that if $\mathrm{rad}([x]) \geq \pi$, then both sets have non-empty intersections with $[x]$. Using standard identities such as $\cos x = \sin(x + \frac{\pi}{2})$, we can obtain interval extensions for all trigonometric functions in a similar fashion.

Exercise 3.1.3 *Compute* $\sin[\frac{\pi}{4}, \frac{3\pi}{4}]$, $\cos[\frac{-\pi}{4}, \frac{3\pi}{2}]$, *and* $\tan[-\frac{\pi}{4}, \frac{\pi}{4}]$.

Exercise 3.1.4 *Find the interval expression for* $tan[x]$, *just as we did for* $sin[x]$.

Exercise 3.1.5 *How would you define* $[x]^{\alpha}$, *for non-integer exponents? Does this definition also work for interval exponents* $[\alpha]$*?.*

For future reference, we define the class of standard functions to be the set

$$\mathfrak{S} = \{a^x, \log_a x, x^{p/q}, \mathrm{abs}\, x, \sin x, \cos x, \tan x, \ldots$$
$$\ldots, \sinh x, \cosh x, \tanh x, \arcsin x, \arccos x, \arctan x\}.$$

By methods similar to those just described, it is possible to extend all standard functions to the interval realm: any $f \in \mathfrak{S}$ has a sharp interval extension F. Again, by sharp, we mean that the interval evaluation $F([x])$ produces the exact range of f over the domain $[x]$:

$$f \in \mathfrak{S} \Rightarrow R(f; [x]) = F([x]).$$

Note that, in particular, this implies that $F([x, x]) = f(x)$, i.e., F and f are identical on $\mathbb{R}$.

Of course, the class of standard functions is too small for most practical applications. We will use them as building blocks for more complicated functions as follows.

Definition 3.1.6 *Any real-valued function expressed as a finite number of standard functions combined with constants, arithmetic operations, and compositions is called an elementary function. The class of elementary functions is denoted by* $\mathfrak{E}$.

Thus an elementary function is recursively defined in terms of its sub-expressions. The leaves of the resulting tree structure (sometimes called a Directed Acyclic Graph - or a DAG for short) are either constants or the variable of the function, see Figure 3.1.

Unfortunately, we cannot obtain sharp extensions of all elementary functions. This is made clear by even very simple examples such as the following:

Example 3.1.7 *Let* $f\colon \mathbb{R} \to \mathbb{R}$ *be defined by* $f(x) = \frac{x}{1+x^2}$. *Its natural interval extension is given by* $F([x]) = \frac{[x]}{1+[x]^2}$. *Thus, for* $[x] = [1, 2]$, *we have*

$$F([x]) = F([1,2]) = \frac{[1,2]}{1 + [1,2]^2} = \frac{[1,2]}{[2,5]} = [\tfrac{1}{5}, 1],$$

whereas $R(f; [1, 2]) = [\frac{2}{5}, \frac{1}{2}]$ *(prove this!). Hence,* $R(f; [1, 2]) \overset{\circ}{\subset} F([1, 2])$.

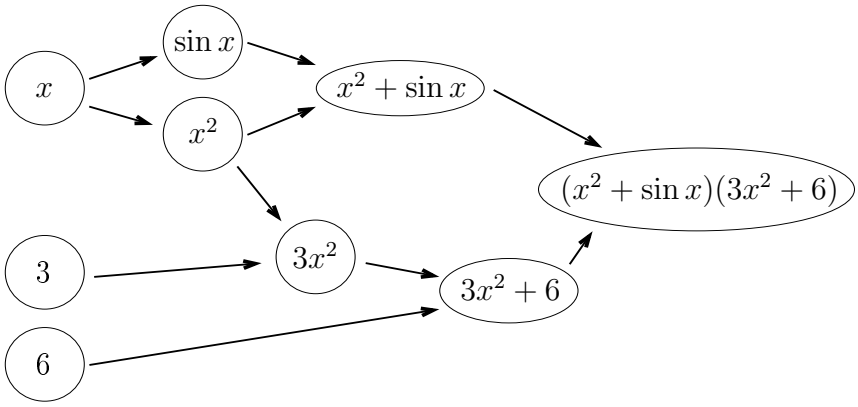


Figure 3.1: The directed acyclic graph for $f(x) = (x^2 + \sin x)(3x^2 + 6)$.

In general, we cannot hope for a sharp extension if the variable x occurs more than once in the explicit representation of f. It should also be pointed out that an elementary function f has infinitely many interval extensions: if F is an extension of f, then so is $F([x]) + [x] - [x]$. Given an explicit representation of an elementary function f, we say that the extension obtained by replacing all occurrences of x by the interval $[x]$ is the *natural* extension. Although the representation of f is immaterial when computing over $\mathbb{R}$, it *does*[2] make a big difference in $\mathbb{IR}$.

Example 3.1.8 *Let $f_1(x) = 1 - x^2$, $f_2(x) = 1 - x \cdot x$, and $f_3(x) = (1 - x)(1 + x)$. These three functions are indistinguishable when evaluated over $\mathbb{R}$. The corresponding natural interval extensions, however, will be distinct functions over $\mathbb{IR}$. They are $F_1([x]) = 1 - [x]^2$, $F_2([x]) = 1 - [x] \times [x]$, and $F_3([x]) = (1 - [x])(1 + [x])$. Note that $F_1 \neq F_2 \neq F_3$ over $\mathbb{IR}$. This is easily seen by evaluating the interval functions at e.g. the interval $[-1, 1]$.*

$$F_1([-1,1]) = 1 - [-1,1]^2 = [1,1] - [0,1] = [0,1],$$

$$F_2([-1,1]) = 1 - [-1,1] \times [-1,1] = [1,1] - [-1,1] = [0,2],$$

$$F_3([-1,1]) = (1 - [-1,1])(1 + [-1,1]) = [0,2] \times [0,2] = [0,4].$$

Somewhat more subtle is the fact that some perfectly valid elementary functions do not admit well-defined natural interval extensions.

Example 3.1.9 *Let $f\colon \mathbb{R} \to \mathbb{R}$ be defined by $f(x) = \frac{\sin \pi x}{x}$. It is a simple exercise to show that $R(f; [-1, 1]) = [0, \pi]$, yet the natural interval extension of f will be undefined for any interval containing the origin due to the (removable) singularity.*

[2] In this sense, computing over $\mathbb{IR}$ and $\mathbb{F}$ is similar: in both situations, the explicit representation of a function is relevant.

Later on, we will learn several ways of how to handle such functions gracefully[3].

Although the elementary functions do not, in general, admit sharp interval-extensions, it is possible to prove that the extensions, when well-defined, are *inclusion isotonic*:

Definition 3.1.10 *Let $[x] \in \mathbb{IR}$. An interval-valued function $F\colon [x] \cap \mathbb{IR} \to \mathbb{IR}$ is inclusion isotonic if, for all $[z] \subseteq [z'] \subseteq [x]$, we have $F([z]) \subseteq F([z'])$.*

We make this statement precise in part (1) of the following theorem:

Theorem 3.1.11 (The fundamental theorem of interval analysis) *Given an elementary function f, and a natural interval-extension F such that $F([x])$ is well-defined for some $[x] \in \mathbb{IR}$, we have*

(1) $[z] \subseteq [z'] \subseteq [x] \Rightarrow F([z]) \subseteq F([z'])$, *(inclusion isotonicity)*

(2) $R(f;[x]) \subseteq F([x])$. *(range enclosure)*

Proof: Recall that an elementary function is recursively defined in terms of its sub-expressions. The theorem clearly holds for rational functions (see Theorem 3.1.2), as well as for the standard functions (since they are sharp). Thus it suffices to prove that if the theorem holds for the elementary functions g_1 and g_2, then it also holds for $g_1 \star g_2$, where $\star \in \{+, -, \times, \div, \circ\}$. We will prove the claim for the composition operator $\circ$; the remaining cases are completely analogous. Since $F([x])$ is well-defined, neither the real-valued function f *nor its sub-expressions* g_i have singularities in their domains induced by the set $[x]$. In particular, g_2 is continuous on any subintervals $[w] \subseteq [w']$ of its domain. This means that $G_2([w])$ and $G_2([w'])$ are compact intervals: $[y]$ and $[y']$, respectively. Therefore, since G_1 and G_2 are inclusion isotonic, we have $[y] \subseteq [y']$, and also

$$G_1 \circ G_2([w]) = G_1([y]) \subseteq G_1([y']) = G_1 \circ G_2([w']).$$

Part (2) now follows immediately by the same argument as given in the proof of Theorem 3.1.2. □

The crucial implication of Theorem 3.1.11 is the fact that we have a means of bounding the range of a function by considering its interval extension. Although the exact range $R(f;[x])$ is virtually impossible to compute, its enclosure $F([x])$ is easy to come by.

Forming the contra-positive of the statement "$y \in R(f;[x]) \Rightarrow y \in F([x])$" produces the very useful "$y \notin F([x]) \Rightarrow y \notin R(f;[x])$". This can be used, e.g., when searching for roots of a function f: if $0 \notin F([x])$, then we know that f has no roots in the domain $[x]$. This can be used for discarding portions of a larger initial search space.

[3] Again, computing over $\mathbb{IR}$ and $\mathbb{F}$ is similar here: in both situations, any naive attempt to evaluate a function at a removable singularity will fail.

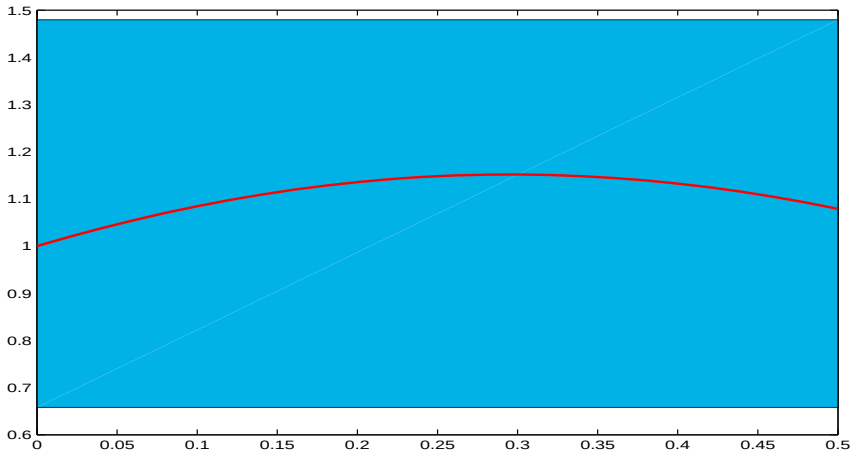


Figure 3.2: The function $f(x) = (\sin x - x^2 + 1)\cos x$ and its interval enclosure over the domain $[0, \frac{1}{2}]$.

Example 3.1.12 *Let $f(x) = (\sin x - x^2 + 1)\cos x$. Prove that f has no roots in the compact interval $[x] = [0, \frac{1}{2}]$.*

The most common way to approach this problem is to find all critical points of f within $[x]$. These are the points for which $f'(x) = 0$. Once they are obtained, we evaluate f at these points, as well as at the endpoints of $[x]$. If all computed function values have the same sign, we can safely conclude that f has no roots in $[x]$. Note, however, that we are simply substituting one root-finding problem ($f(x) = 0$) for another ($f'(x) = 0$). It may very well be the case that f' is more complicated than f, in which case we will have gained nothing.

Using interval techniques, however, we can obtain a proof in just one evaluation of the interval extension of f:

$$\begin{aligned} F([0,\tfrac{1}{2}]) &= (\sin[0,\tfrac{1}{2}] - [0,\tfrac{1}{2}]^2 + 1)\cos[0,\tfrac{1}{2}] = ([0,\sin\tfrac{1}{2}] - [0,\tfrac{1}{4}] + 1) \times [\cos\tfrac{1}{2}, 1] \\ &= ([0,\sin\tfrac{1}{2}] + [\tfrac{3}{4}, 1]) \times [\cos\tfrac{1}{2}, 1] = [\tfrac{3}{4}, 1 + \sin\tfrac{1}{2}] \times [\cos\tfrac{1}{2}, 1] \\ &= [\tfrac{3}{4}\cos\tfrac{1}{2}, 1 + \sin\tfrac{1}{2}] \subseteq [0.65818, 1.4795]. \end{aligned}$$

Now, since $0 \notin F([0, \frac{1}{2}])$, it follows that $0 \notin R(f; [0, \frac{1}{2}])$, so f has no roots in the interval $[0, \frac{1}{2}]$, see Figure 3.2.

Here we were fortunate: with just one interval evaluation, the range could be enclosed sufficiently tight for our needs. Occasionally, we will not get *any* information with just one interval evaluation. This happens when $F([x])$ is not well-defined, which can occur even though $R(f; [x])$ is a compact interval. The following example illustrates that when this happens, we may invoke part (1) of the Theorem 3.1.11 to obtain a more satisfactory result.

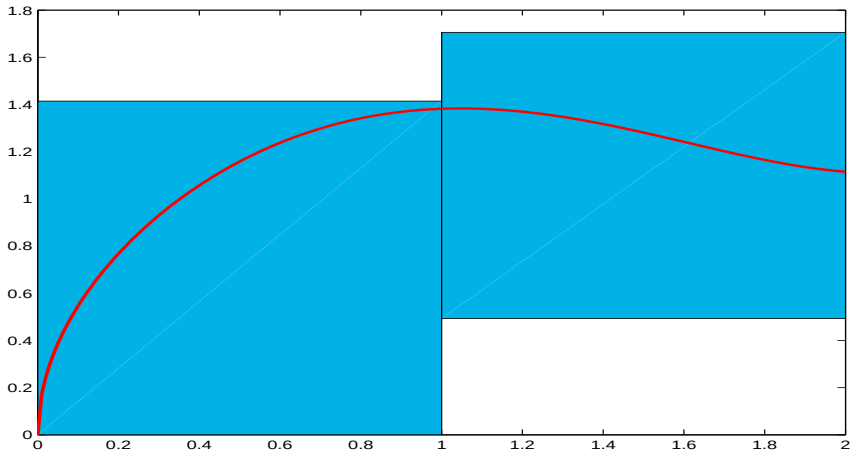


Figure 3.3: The function $f(x) = \sqrt{x + \sin 2x}$ and its interval enclosure over the domain $[0, 2]$.

Example 3.1.13 *Find an enclosure of $R(f; [0, 2])$, where $f(x) = \sqrt{x + \sin 2x}$. This time, a single evaluation of the interval extension of f will not suffice:*

$$\begin{aligned} R(f; [0, 2]) \subseteq F([0, 2]) &= \sqrt{[0, 2] + \sin(2 \times [0, 2])} = \sqrt{[0, 2] + \sin[0, 4]} \\ &= \sqrt{[0, 2] + [\sin 4, \sin \pi/2]} = \sqrt{[0, 2] + [\sin 4, 1]} \\ &= \sqrt{[\sin 4, 3]}. \end{aligned}$$

Note that the final expression cannot be properly evaluated since $\sin 4$ is negative[4]*. If we bisect the original domain $[0, 2] = [0, 1] \cup [1, 2]$, we can use the inclusion isotonicity of F to obtain*

$$\begin{aligned} R(f; [0, 2]) &= R(f; [0, 1]) \cup R(f; [1, 2]) \subseteq F([0, 1]) \cup F([1, 2]) \\ &= \sqrt{[0, 1] + \sin(2 \times [0, 1])} \cup \sqrt{[1, 2] + \sin(2 \times [1, 2])} \\ &= \sqrt{[0, 1] + \sin[0, 2]} \cup \sqrt{[1, 2] + \sin[2, 4]} \\ &= \sqrt{[0, 1] + [\sin 0, \sin \tfrac{\pi}{2}]} \cup \sqrt{[1, 2] + [\sin 4, \sin 2]} \\ &= \sqrt{[0, 1] + [0, 1]} \cup \sqrt{[1 + \sin 4, 2 + \sin 2]} \\ &= [0, \sqrt{2}] \cup [\sqrt{1 + \sin 4}, \sqrt{2 + \sin 2}] \\ &= [0, \sqrt{2 + \sin 2}] \subseteq [0, 1.7057]. \end{aligned}$$

This produces a valid (non-sharp) enclosure of $R(f; [0, 2])$, see Figure 3.3.

Of course, the enclosure $F([x])$ may grossly overestimate $R(f; [x])$. If f is sufficiently regular, however, this overestimation can be made arbitrarily small by subdividing $[x]$ into many smaller pieces, evaluating F over each smaller piece, and then taking the union of all resulting sets.

[4] One can also propose the interpretation $\sqrt{[x]} = \sqrt{[x] \cap [0, +\infty]}$, which resolves this issue. See e.g. [PC06] and [HW04], where so called cset theory is introduced.

Definition 3.1.14 *A function $f\colon D \to \mathbb{R}$ is Lipschitz if there exists a positive constant K such that, for all $x, y \in D$, we have $|f(x) - f(y)| \leq K|x - y|$. The number K is called a Lipschitz constant of f over D.*

Thus a Lipschitz function f is necessarily continuous, but it need not be differentiable. If f happens to be differentiable, then the modulus of the derivative is always bounded by the Lipschitz constant K.

We now define $\mathfrak{E}_{\mathfrak{L}}$ to be the set of all elementary functions whose sub-expressions are all Lipschitz:

$$\mathfrak{E}_{\mathfrak{L}} = \{f \in \mathfrak{E}\colon \text{each sub-expression of } f \text{ is Lipschitz}\}.$$

Theorem 3.1.15 (Range enclosure) *Consider $f\colon I \to \mathbb{R}$ with $f \in \mathfrak{E}_{\mathfrak{L}}$, and let F be an inclusion isotonic interval extension of f such that $F([x])$ is well-defined for some $[x] \subseteq I$. Then there exists a positive real number K, depending on F and $[x]$, such that, if $[x] = \cup_{i=1}^{k}[x^{(i)}]$, then*

$$R(f;[x]) \subseteq \bigcup_{i=1}^{k} F([x^{(i)}]) \subseteq F([x])$$

and

$$\operatorname{rad}\left(\bigcup_{i=1}^{k} F([x^{(i)}])\right) \leq \operatorname{rad}\big(R(f;[x])\big) + K \max_{i=1,\ldots,k} \operatorname{rad}\left([x^{(i)}]\right).$$

Proof: We start with the first inclusion. By the the inclusion isotonic property, it follows that by splitting the box $[x]$ into smaller pieces $[x] = \cup_{i=1}^{k}[x^{(i)}]$, we have

$$R(f;[x]) = R(f;\cup_{i=1}^{k}[x^{(i)}]) = \bigcup_{i=1}^{k} R(f;[x^{(i)}]) \subseteq \bigcup_{i=1}^{k} F([x^{(i)}]) \subseteq F(\cup_{i=1}^{k}[x^{(i)}]) = F([x]).$$

To prove the second statement, we must show that if $\tilde{\xi} \in \cup_{i=1}^{k} F([x^{(i)}])$, then there exists $\xi \in R(f;[x])$ such that $|\tilde{\xi} - \xi| \leq K \max_{i=1,\ldots,k} \operatorname{rad}\left([x^{(i)}]\right)$. We will prove the more precise statement that if $[w] \subseteq [x]$ and $\tilde{\xi} \in F([w])$, then for all $\xi \in R(f;[w])$ we have $|\tilde{\xi} - \xi| \leq K \operatorname{rad}([w])$. This clearly implies the second part of the theorem.

We first note that the statement is trivially valid for constants and standard functions, since they return sharp enclosures (which are thus bounded). Therefore it suffices to show that, if the statement is true for two connecting branches g_1 and g_2 of the recursive tree for f, then it also holds for $g_1 \star g_2$, where $\star \in \{+, -, \times, \div, \circ\}$. In a sub-expression like $\sin x^2$, we interpret sin and x^2 as connecting branches. As in the proof of Theorem 3.1.11, we will only prove the claim for the composition operator $\circ$, as the remaining cases are very similar.

Since f is an elementary function, g_1 and g_2 are also elementary, so their interval extensions G_1 and G_2 are inclusion isotonic by Theorem 3.1.11. Furthermore, since

$f \in \mathfrak{E}_{\mathfrak{L}}$, g_1 and g_2 are Lipschitz on their domains, and since $F([x])$ is well-defined, the extensions G_1 and G_2 are also well-defined on their domains ($[w_1]$ resp. $[w_2]$) induced by $[x]$.

Our inductive assumptions are

$$[v] \subseteq [w_i],\ \tilde{z} \in G_i([v]), \text{ and } z \in R(g_i;[v]) \Rightarrow |\tilde{z} - z| \leq K_i \mathrm{rad}\,([v])\,. \qquad (i = 1, 2)$$

From part (2) of Theorem 3.1.11, we have (for all $[v] \subseteq [w_2]$)

$$R\,(g_1 \circ g_2; [v]) = R\,(g_1; R\,(g_2; [v])) \subseteq R\,(g_1; G_2\,([v]))\,.$$

Now, if $z \in R\,(g_1 \circ g_2; [v])$, then there exists $u \in R\,(g_2; [v])$ such that $z = g_1(u)$. Note that $u \in G_2\,([v])$. Therefore, if $\tilde{z} \in G_1 \circ G_2([v]) = G_1\,(G_2\,([v]))$, then by our assumptions on g_1 and G_1, it follows that

$$|z - \tilde{z}| \leq K_1 \mathrm{rad}\,(G_2\,([v]))\,.$$

Now, by our assumptions on g_2 and G_2, it follows that

$$\mathrm{rad}\,(G_2\,([v])) \leq \mathrm{rad}\,(R\,(g_2; [v])) + K_2 \mathrm{rad}\,([v]) \leq K_3 \mathrm{rad}\,([v]) + K_2 \mathrm{rad}\,([v])\,,$$

where we used the Lipschitz property of g_2 in the final estimate. Combining these two inequalities yields

$$|z - \tilde{z}| \leq K_1(K_3 + K_2)\mathrm{rad}\,([v])\,.$$

Since f is elementary, its recursive tree is finite, and thus the accumulated constants from successive estimates as above will be finite. This finishes the proof. □

In essence, the second part of Theorem 3.1.15 says that, if the listed conditions are satisfied, then the overestimation tends to zero no slower than linearly as the domain shrinks:

$$\mathrm{rad}([x]) = \mathcal{O}(\varepsilon) \Rightarrow d\big(R\,(f; [x])\,, F\,([x])\,\big) = \mathcal{O}(\varepsilon),$$

where $d(\cdot, \cdot)$ is the Hausdorff distance, as defined in (2.1). Since Lipschitz functions satisfy $\mathrm{rad}\big(R(f; [x])\big) = \mathcal{O}(\mathrm{rad}([x]))$, it also follows that

$$\mathrm{rad}([x]) = \mathcal{O}(\varepsilon) \Rightarrow \mathrm{rad}\big(F\,([x])\,\big) = \mathcal{O}(\varepsilon),$$

i.e., the width of the enclosure scales at least linearly with ε.

Exercise 3.1.16 *Prove that, if f is Lipschitz with Lipschitz constant K, then*

$$\mathrm{rad}\big(R(f; [x])\big) \leq K \mathrm{rad}([x]).$$

Example 3.1.17 *Let $f(x) = (\sin x - x^2 + 1)\cos x$ be as in Example 3.1.12, and consider the domain $[0, \varepsilon]$, where ε is small. By a direct computation we have*

$$\begin{aligned} R\left(f; [0,\varepsilon]\right) &= [1, (\sin\varepsilon - \varepsilon^2 + 1)\cos\varepsilon], \\ F\left([0,\varepsilon]\right) &= [(1-\varepsilon^2)\cos\varepsilon, 1 + \sin\varepsilon]. \end{aligned}$$

The overestimation is therefore given by the distance between the two intervals:

$$\begin{aligned} &d\big(R\left(f; [0,\varepsilon]\right), F\left([0,\varepsilon]\right)\big) = \\ &\quad = \max\left\{\left|1 - (1-\varepsilon^2)\cos\varepsilon\right|, \left|(\sin\varepsilon - \varepsilon^2 + 1)\cos\varepsilon - (1+\sin\varepsilon)\right|\right\} \\ &\quad = \tfrac{3}{2}\varepsilon^2 + \mathcal{O}(\varepsilon^3) = \mathcal{O}(\varepsilon^2) \end{aligned}$$

Note that this is actually better than claimed by the theorem: the overestimation is quadratic in ε rather than being linear.

If we consider functions that are not Lipschitz, then the results of the theorem may not hold.

Example 3.1.18 *Let $f(x) = \sqrt{x + \sin 2x}$ be as in Example 3.1.13, and consider the domain $[0, \varepsilon]$, where ε is small. Again, by a direct computation we have*

$$R\left(f; [0,\varepsilon]\right) = [0, \sqrt{\varepsilon + \sin 2\varepsilon}] = F\left([0,\varepsilon]\right).$$

In this particular case, there is no overestimation at all! Nevertheless, if we measure the width of the enclosure, we have

$$\operatorname{rad}\big(F\left([0,\varepsilon]\right)\big) = \frac{1}{2}\sqrt{\varepsilon + \sin 2\varepsilon} = \mathcal{O}(\sqrt{\varepsilon}) \neq \mathcal{O}(\varepsilon).$$

This "slower than linear" scaling is due to the fact that f' has a singularity at the origin.

As a consequence of Theorem 3.1.15, we can generate tight enclosures of the graph of a given function. By partitioning the domain into sufficiently small subintervals, we can obtain the desired accuracy, i.e., the overestimation can be made as small as we wish. In Figure 3.4, this is illustrated for the function $f(x) = \cos^3 x + \sin x$ over the domain $[-5, 5]$. By setting the tolerance in the range width (i.e. the height of the boxes) to 2, 1, 0.5, and 0.25, we get 4, 8, 22, and 40 enclosing boxes, respectively.

3.2 Centered forms

In some situations, we can arrange for the overestimation of the exact range to tend to zero faster than linearly as the domain shrinks. This is possible if the function f satisfies the Mean Value Theorem.

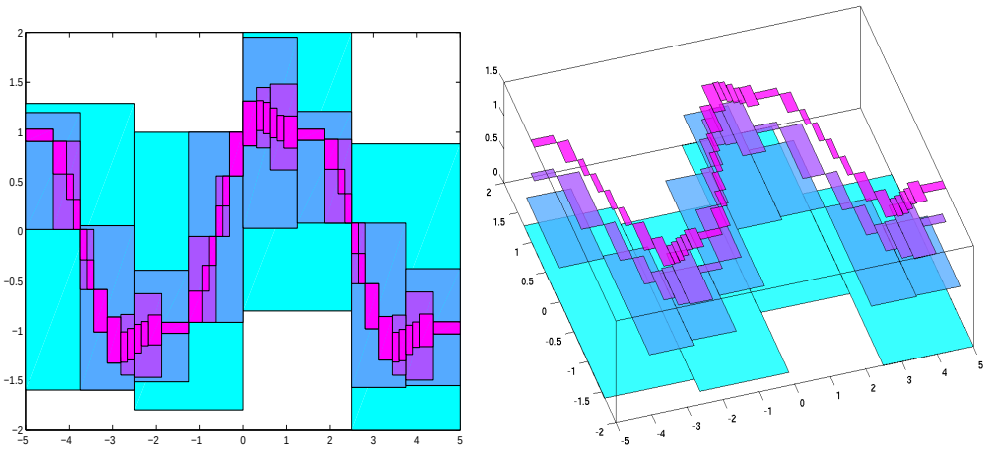


Figure 3.4: Two views of the function $f(x) = \cos^3 x + \sin x$ and its successively tighter interval enclosures over the domain $[-5, 5]$

Theorem 3.2.1 (The Mean Value Theorem) *If f is continuous on $[a, b]$, and differentiable on (a, b), then there is a number ζ in (a, b) such that*

$$f'(\zeta) = \frac{f(b) - f(a)}{b - a}.$$

For the remainder of this section, we will only consider functions satisfying the assumptions of the Mean Value Theorem.

Thus, given $f\colon [x] \to \mathbb{R}$, the Mean Value Theorem states that, if x and c are points in $[x]$, then there is a ζ between x and c (and thus also in $[x]$) such that

$$f(x) = f(c) + f'(\zeta)(x - c).$$

Now suppose that we have an interval extension F' for the derivative f'. It then follows that

$$\begin{aligned} f(x) &= f(c) + f'(\zeta)(x - c) \in f(c) + F'([x])(x - c) \\ &\subseteq f(c) + F'([x])([x] - c), \end{aligned}$$

where the last expression is independent of both x and ζ. Thus, for all x and c in the domain $[x]$, we have

$$R(f; [x]) \subseteq f(c) + F'([x])([x] - c) \stackrel{\text{def}}{=} F([x]; c).$$

The interval function $F([x]; c)$ is called a *centered form*[5]. The most popular choice is to take $c = \text{mid}([x])$, which produces the *mean-value form*,

$$F_m([x]) \stackrel{\text{def}}{=} F([x]; \text{mid}([x])).$$

[5]When working in finite precision, it is crucial to compute $F([c, c])$ rather than $f(c)$ in order to take all rounding errors into account.

Using the shorthand notations $m = \text{mid}([x])$ and $r = \text{rad}([x])$, we have

$$F_m([x]) = f(m) + F'([x])([x] - m) = f(m) + F'([x])[-r, r].$$

One of the benefits of using centered forms is that they provide us with *explicit* bounds on the overestimation of the exact range, as opposed to Theorem 3.1.15.

Theorem 3.2.2 (Centered form enclosure) *Consider an elementary function $f : I \to \mathbb{R}$ satisfying the Mean Value Theorem, and let F' be an interval extension of f' such that $F'([x])$ is well-defined for some $[x] \subseteq I$. Then, if c belongs to $[x]$, we have*

$$R(f; [x]) \subseteq F([x]; c)$$

and

$$\text{rad}\big(F([x]; c)\big) \leq \text{rad}\big(R(f; [x])\big) + 4\text{rad}\big(F'([x])\big)\text{rad}\big([x]\big).$$

Proof: We have already proved the inclusion $R(f; [x]) \subseteq F([x]; c)$, so we need only concentrate our efforts toward proving the bound on the overestimation.

Let y be any element of $F([x]; c)$. Then there is a $u \in F'([x])$ and a $v \in [x] - c$ such that $y = f(c) + uv$. Now, $v = x - c$ for some $x \in [x]$, so

$$y = f(c) + u(x - c). \tag{3.1}$$

By the Mean Value Theorem, we have $f(x) = f(c) + f'(\zeta)(x - c)$ for some $\zeta \in [x]$, i.e.

$$f(c) = f(x) - f'(\zeta)(x - c) \tag{3.2}$$

Combining equations (3.1) and (3.2) produces

$$\begin{aligned} y &= f(x) - f'(\zeta)(x - c) + u(x - c) = f(x) + (u - f'(\zeta))(x - c) \\ &\in R(f; [x]) + \big(F'([x]) - F'([x])\big)([x] - c). \end{aligned}$$

Since y was an arbitrary element of $F([x]; c)$, it follows that the overestimation is bounded by the size of the set

$$\big(F'([x]) - F'([x])\big)([x] - c). \tag{3.3}$$

Note that the first factor of (3.3) is an interval of the form $[a] = \lambda \cdot [-1, 1]$. If $[b]$ is an arbitrary interval, we have a particularly elegant formula for the product:

$$[a] \times [b] = \lambda \cdot \text{mag}([b]) \cdot [-1, 1].$$

Setting $[a] = F'([x]) - F'([x])$ and $[b] = [x] - c$ produces $\lambda = 2\text{rad}\big(F'([x])\big)$ and $\text{mag}([b]) \leq 2\text{rad}\big([x]\big)$, i.e.,

$$\big(F'([x]) - F'([x])\big)([x] - c) \subseteq 4\text{rad}\big(F'([x])\big)\text{rad}\big([x]\big) \cdot [-1, 1].$$

Thus we have established the desired bound

$$\mathrm{rad}\Big(\big(F'([x]) - F'([x])\big)([x]-c)\Big) \le 4\mathrm{rad}\big(F'([x])\big)\mathrm{rad}\big([x]\big),$$

which concludes the proof. □

We remark that the bound can be improved by a factor two if the mean-value form is used, rather than a general centered form. This is because we then have $[b] = [x] - \mathrm{mid}\big([x]\big) = \mathrm{rad}\big([x]\big)\cdot[-1,1]$, which gives the improved bound $\mathrm{mag}([b]) = \mathrm{rad}\big([x]\big)$.

Exercise 3.2.3 *Prove that a centered form $F([x],c)$ is not inclusion isotonic unless $c = \mathrm{mid}([x])$.*

It is clear from Theorem 3.2.2 that we obtain a very tight enclosure of $R(f;[x])$ when both $F'([x])$ and $[x]$ are narrow intervals. If f' satisfies the conditions of Theorem 3.1.15, then $\mathrm{rad}\big([x]\big) = \mathcal{O}(\varepsilon) \Rightarrow \mathrm{rad}\big(F'([x])\big) = \mathcal{O}(\varepsilon)$, and the overestimation is roughly of the order $\mathcal{O}(\varepsilon^2)$. Nevertheless, it does not follow that centered forms always produce better enclosures than a single interval evaluation.

Example 3.2.4 *Let $f(x) = \frac{x^2+1}{x}$. Then $f'(x) = \frac{x^2-1}{x^2}$, so $F'([x])$ is well-defined as long as $0 \notin [x]$. Let $[x] = [1,2]$. Then the interval evaluation gives*

$$F([1,2]) = \frac{[1,2]^2+1}{[1,2]} = \frac{[2,5]}{[1,2]} = [1,5],$$

whereas the mean value form produces

$$\begin{aligned} F_m([1,2]) &= f(\tfrac{3}{2}) + F'([1,2])[-\tfrac{1}{2},\tfrac{1}{2}] \\ &= \tfrac{13}{6} + \frac{[1,2]^2-1}{[1,2]^2}[-\tfrac{1}{2},\tfrac{1}{2}] = \tfrac{13}{6} + \frac{[0,3]}{[1,4]}[-\tfrac{1}{2},\tfrac{1}{2}] \\ &= \tfrac{13}{6} + [0,3][-\tfrac{1}{2},\tfrac{1}{2}] = \tfrac{13}{6} + [-\tfrac{3}{2},\tfrac{3}{2}] = [\tfrac{2}{3},\tfrac{11}{3}]. \end{aligned}$$

Thus $\mathrm{rad}\big(F_m([x])\big) = \frac{3}{2} < 2 = \mathrm{rad}\big(F([x])\big)$, so the mean-value form gives the most narrow enclosure. In spite of this, the interval evaluation produces a better lower bound for the exact range. Intersecting both enclosures produces the even tighter enclosure:

$$R(f;[1,2]) \subseteq F([1,2]) \cap F_m([1,2]) = [1,\tfrac{11}{3}].$$

Example 3.2.5 *Let $f(x) = \sin x^2$. Then $f'(x) = 2x\cos x^2$, so if we introduce $[x] = m + [-r,r]$, then we have*

$$F_m([x]) = f(m) + F'([x])[-r,r] = \sin m^2 + 2[x]\cos\big([x]^2\big)[-r,r].$$

For $[x] = [-\varepsilon, \varepsilon]$, *we have* $m = 0$ *and* $r = \varepsilon$, *which gives*

$$\begin{aligned} F_m([-\varepsilon,\varepsilon]) &= \sin 0^2 + 2[-\varepsilon,\varepsilon]\cos\big([0,\varepsilon^2]\big)[-\varepsilon,\varepsilon] \\ &= 2[-\varepsilon^2,\varepsilon^2][\cos\varepsilon^2, 1] = 2[-\varepsilon^2,\varepsilon^2]. \end{aligned}$$

The interval evaluation, however, produces the sharp enclosure:

$$F([-\varepsilon,\varepsilon]) = \sin[-\varepsilon,\varepsilon]^2 = \sin[0,\varepsilon^2] = [0,\sin\varepsilon^2] = R(f;[-\varepsilon,\varepsilon]).$$

In this particular case, the mean-value form produces an enclosure that is more than four times wider than the exact range:

$$\operatorname{rad}\big(F([x])\big) \le \tfrac{1}{4}\operatorname{rad}\big(F_m([x])\big).$$

3.3 Monotonicity

When computing a centered form, the quantity $F'([x])$ must be calculated. If it happens that $0 \notin F'([x])$, then f is monotone on $[x]$, and a sharp result can easily be obtained:

$$R(f;[x]) = \begin{cases} [f(\underline{x}), f(\overline{x})] & \text{if } \min\{y \in F'([x])\} \ge 0, \\ [f(\overline{x}), f(\underline{x})] & \text{if } \max\{y \in F'([x])\} \le 0. \end{cases}$$

Example 3.3.1 *Let us return to Example 3.2.4, where* $f(x) = \frac{x^2+1}{x}$ *is considered on the domain* $[x] = [1,2]$. *Then* $f'(x) = \frac{x^2-1}{x^2}$, *so*

$$F'([1,2]) = \frac{[1,2]^2 - 1}{[1,2]^2} = \frac{[1,4]-1}{[1,4]} = \frac{[0,3]}{[1,4]} = [0,3].$$

This means that f *is non-decreasing on* $[1,2]$, *so the exact range is given by evaluating the endpoints:*

$$R(f;[1,2]) = [f(1), f(2)] = [2, \tfrac{5}{2}].$$

If we compare the widths resulting from the various approaches, we have

$$\operatorname{rad}\big(R(f;[1,2])\big) = \tfrac{1}{2} < \operatorname{rad}\big(F_m([x])\big) = \tfrac{3}{2} < \operatorname{rad}\big(F([x])\big) = 2.$$

In Chapter 4, we will present a very powerful technique allowing us to generate graph enclosures of any desired order. This can be used to obtain very tight enclosures of the range.

3.4 Computer Lab II

Problem 1. Write a small set of routines supporting interval versions of the standard functions e^x, $\log x$, x^n ($n \in \mathbb{Z}$), x^α ($\alpha \in \mathbb{R} \setminus \mathbb{Z}$), $\sin x$, $\cos x$, and $\tan x$. If you have access to directed rounding, use it. You may assume the the floating point implementation of each standard function returns the floating point nearest the exact value.

Problem 2. Merge the routines from the previous problem with the routines from Problem 6 in Computer Lab I. Use this to compute $f(1+2^{-k}[-1,1])$ for $k = 0, \ldots, 10$, where

$$f(x) = e^{\sin e^{\cos x + 2x^5}},$$

or in a more functional form: `f(x) = exp(sin(exp(cos(x) + 2*pow(x,5))))`.

Problem 3. Let f be as in Problem 2 above. Compute (by hand?) the formal expression for $f'(x)$, and implement the mean-value form $F_m([x])$. Use this to compute $f(1+2^{-k}[-1,1])$ for $k = 0, \ldots, 10$. Compare the quality of the enclosures with those from Problem 2.

Chapter 4

Automatic differentiation

In this chapter, we will present an elegant and effective technique for automatically generating n-th order derivatives of a given function. This will open the path to a series of powerful interval methods, almost completely removing the main obstruction of overestimation seen in the previous chapters. As a nice bi-product, the technique also relieves us from the teadious and error-prone task of hand-coding functional expressions for derivatives. A thorough treatment of the topic at hand can be found in e.g. [Gr00].

4.1 First-order derivatives

The aim of this section is to produce the value $f'(x_0)$, given the point x_0 and a formula for f. This is usually achieved by either computing an approximation of $f'(x_0)$ via one of the divided differences

$$\begin{aligned} \Delta_h^+(f, x_0) &\stackrel{\text{def}}{=} \frac{f(x_0 + h) - f(x_0)}{h}, \\ \Delta_h^-(f, x_0) &\stackrel{\text{def}}{=} \frac{f(x_0) - f(x_0 - h)}{h}, \\ \Delta_h^\pm(f, x_0) &\stackrel{\text{def}}{=} \frac{f(x_0 + h) - f(x_0 - h)}{2h}, \end{aligned} \tag{4.1}$$

or by generating the exact symbolic formula for f', and then evaluating it at x_0. Both approaches have serious disadvantages. The former is prone to producing gross errors due to both the discretization, and the finite precision computations. The latter approach is both memory- and time-consuming, and many functions f simply cannot be handled in a reasonably short period of time.

We will, much like with the interval arithmetic, start from scratch, and construct a *differentiation arithmetic*. Focussing initially on the real-valued setting, we will perform all calculations with ordered pairs of real numbers

$$\vec{u} = (u, u'),$$

where u denotes the value of the function $u\colon \mathbb{R} \to \mathbb{R}$ evaluated at the point x_0, and where u' denotes the value $u'(x_0)$. The basic arithmetic rules are

$$\begin{aligned} \vec{u} + \vec{v} &= (u + v, u' + v') \\ \vec{u} - \vec{v} &= (u - v, u' - v') \\ \vec{u} \times \vec{v} &= (uv, uv' + u'v) \\ \vec{u} \div \vec{v} &= (u/v, (u' - (u/v)v')/v), \end{aligned} \tag{4.2}$$

where we demand that $v \neq 0$ when dividing. The rule for division is derived from

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2} = \frac{u' - (u/v)v'}{v}.$$

Note that the quantity (u/v) need only be computed once, although it appears twice in the formula for $\vec{u} \div \vec{v}$.

In order to be able to compute with this new arithmetic, we need to know how constants and the independent variable x are treated. Following the usual rules of differentiation, we define

$$\vec{x} = (x, 1) \qquad \text{and} \qquad \vec{c} = (c, 0). \tag{4.3}$$

We are now ready to reap the first fruits of our labor. Let f be a rational function, and replace all occurances of the variable x with $\vec{x}$, each constant c_i with $\vec{c}_i$, and all arithmetic operations with their counterparts from (4.2). This produces the new function $\vec{f}$ which, upon evaluation at $(x_0, 1)$, produces the ordered pair $(f(x_0), f'(x_0))$.

Example 4.1.1 *Let $f(x) = \frac{(x+1)(x-2)}{x+3}$. We wish to compute the values of $f(3)$ and $f'(3)$. It is easy to see that $f(3) = 2/3$. The value of $f'(3)$, however, is not immediate. Applying the techniques of differentiation arithmetic, we define*

$$\vec{f}(\vec{x}) = \frac{(\vec{x} + \vec{1})(\vec{x} - \vec{2})}{\vec{x} + \vec{3}} = \frac{\big((x,1) + (1,0)\big) \times \big((x,1) - (2,0)\big)}{(x,1) + (3,0)}.$$

Inserting the value $\vec{x} = (3,1)$ into $\vec{f}$ produces

$$\begin{aligned} \vec{f}(3,1) &= \frac{\big((3,1) + (1,0)\big) \times \big((3,1) - (2,0)\big)}{(3,1) + (3,0)} \\ &= \frac{(4,1) \times (1,1)}{(6,1)} = \frac{(4,5)}{(6,1)} = (\frac{2}{3}, \frac{13}{18}). \end{aligned}$$

From this calculation it follows that $f(3) = 2/3$ (which we already knew) and $f'(3) = 13/18$. Note that we never used the expression for f'.

If we use the different (but equivalent) representation $f(x) = x - \frac{4x+2}{x+3}$*, the same procedure as above yields*

$$\begin{aligned}\vec{f}(3,1) &= (3,1) - \frac{(4,0)\times(3,1)+(2,0)}{(3,1)+(3,0)} \\ &= (3,1) - \frac{(12,4)+(2,0)}{(6,1)} = (3,1) - \frac{(14,4)}{(6,1)} \\ &= (3,1) - (\frac{7}{3}, \frac{5}{18}) = (\frac{2}{3}, \frac{13}{18}).\end{aligned}$$

Thus we arrive at the same result by a completely different route.

Exercise 4.1.2 *Write a program module that implements the data type* `autodiff` *defined by the rules (4.2) and (4.3). Use the module to compute the derivatives of more complicated rational functions that the ones used in Examples 4.1.1 and 4.1.4.*

Exercise 4.1.3 *It is straight-forward to extend the forward difference* $\Delta_h^+(f, x_0)$ *appearing in 4.1 to the interval version*

$$\Delta_h^+(F, [x_0]) \stackrel{\text{def}}{=} \frac{F([x_0]+h) - F([x_0])}{h}.$$

Explain why this interval function may not enclose the derivative $f'(x_0)$*.*

We now want to extend these techniques to elementary functions. This is achieved by implementing the chain rule $(g\circ u)'(x) = u'(x)(g'\circ u)(x)$ in the second component of the ordered pairs. We thus define the rule

$$\vec{g}(\vec{u}) = \vec{g}\big((u, \vec{u})\big) = \big(g(u), u'g'(u)\big). \tag{4.4}$$

To give some examples, we define extensions of the following standard functions:

$$\begin{array}{lcll} \sin\vec{u} & = & \sin(u,u') & = & (\sin u, u'\cos u) & \\ \cos\vec{u} & = & \cos(u,u') & = & (\cos u, -u'\sin u) & \\ e^{\vec{u}} & = & e^{(u,u')} & = & (e^u, u'e^u) & \\ \log\vec{u} & = & \log(u,u') & = & (\log u, u'/u) & (u>0) \\ \vec{u}^{\alpha} & = & (u,u')^{\alpha} & = & (u^{\alpha}, u'\alpha u^{\alpha-1}) & (\alpha\neq 0) \\ |\vec{u}| & = & |(u,u')| & = & (|u|, u'\mathrm{sign}(u)) & (u\neq 0). \end{array} \tag{4.5}$$

This list can be complemented by more functions, such as $\tan x$, $\arcsin x$, etc. We can then automatically generate the first derivative of any elementary function by combining the rules (4.2), (4.4), and (4.5).

Example 4.1.4 *Let* $f(x) = (1 + x + e^x)\sin x$*, and suppose we want to compute* $f'(0)$*. As in Example 4.1.1, we define the extended function*

$$\vec{f}(\vec{x}) = (\vec{1} + \vec{x} + e^{\vec{x}})\sin\vec{x},$$

and evaluate it at $\vec{x} = (0, 1)$. *This gives*

$$\begin{aligned}\vec{f}(0,1) &= \big((1,0) + (0,1) + e^{(0,1)}\big)\sin(0,1) \\ &= \big((1,1) + (e^0, e^0)\big)(\sin 0, \cos 0) = (2,2)(0,1) = (0,2).\end{aligned}$$

From this simple calculation, it follows that $f(0) = 0$, *and* $f'(0) = 2$.

Exercise 4.1.5 *Extend your module using (4.5) and (4.4) so it handles elementary functions too. Use the module to compute the derivatives of some non-trivial elementary functions.*

Note that it is possible to combine interval arithmetic with differentiation arithmetic. We then compute with ordered pairs of intervals $\vec{u} = ([u], [u'])$. The result of such a computation is a pair of intervals enclosing the range of the function and its derivative, respectively.

Example 4.1.6 *Let* $f(x) = 1 + \sin(2x)$, *and suppose that we want to compute the enclosure* $F'([0, \frac{\pi}{4}])$. *As in the previous examples, we define the extended function*

$$\vec{F}([\vec{x}]) = \vec{1} + \sin(2[\vec{x}]),$$

except that we now evaluate $\vec{F}$ *at the interval pair* $[\vec{x}] = ([0, \frac{\pi}{4}], 1)$. *This gives*

$$\begin{aligned}\vec{F}([0,\tfrac{\pi}{4}],1) &= (1,0) + \sin\big((2,0)([0,\tfrac{\pi}{4}],1)\big) = (1,0) + \sin([0,\tfrac{\pi}{2}],2) \\ &= (1,0) + (\sin[0,\tfrac{\pi}{2}], 2\cos[0,\tfrac{\pi}{2}]) = (1,0) + ([0,1], 2[0,1]) \\ &= (1,0) + ([0,1],[0,2]) = ([1,2],[0,2])\end{aligned}$$

From this calculation, it follows that $F([0, \frac{\pi}{4}]) = [1, 2]$, *and* $F'([0, \frac{\pi}{4}]) = [0, 2]$. *It just happens (due to monotonicity) that both enclosures are sharp:* $R(f; [0, \frac{\pi}{4}]) = [1, 2]$ *and* $R(f'; [0, \frac{\pi}{4}]) = [0, 2]$.

Exercise 4.1.7 *Write a short program that links your interval module with the automatic differentiation module just created. Given an elementary function* f *and an interval* $[x]$, *the program should return enclosures of* $R(f; [x])$ *and* $R(f'; [x])$.

4.2 Higher-order derivatives

It should come as no surprise that the techniques presented in the previous section can be generalized to produce derivatives of higher order than one. As an example, for second order derivatives, we must compute with triples of numbers $\vec{u} = (u, u', u'')$, and the arithmetic rules corresponding to (4.2) generalize to

$$\begin{aligned}\vec{u} + \vec{v} &= (u + v, u' + v', u'' + v'') \\ \vec{u} - \vec{v} &= (u - v, u' - v', u'' - v'') \\ \vec{u} \times \vec{v} &= (uv, uv' + u'v, uv'' + 2u'v' + u''v) \\ \vec{u} \div \vec{v} &= (u/v, (u' - (u/v)v')/v, (u'' - 2(u/v)'v' - (u/v)v'')/v),\end{aligned} \tag{4.6}$$

where we, as before, demand that $v \neq 0$ when dividing. Extending the second-order rules to the standard functions is straight-forward, but tedious[1].

A more effective (and perhaps less error-prone) approach to high-order automatic differentiation is obtained through the calculus of Taylor series (see e.g. [Mo66], [Mo79], [Ab88], or [Ab98]). Given a real-valued function $f \in C^\infty$, its Taylor expansion at x_0 is

$$f(x) = f_0 + f_1(x - x_0) + \cdots + f_k(x - x_0)^k + \ldots,$$

where the Taylor coefficients are given by $f_k = f_k(x_0) = f^{(k)}(x_0)/k!$. Here, we are using the common notation for derivatives: $f^{(k)} = \frac{d^k f}{dx^k}$. Now, since the Taylor coefficients are just rescaled derivatives, it follows that computing the derivatives of a function is equivalent to computing its Taylor series.

It is clear that adding the Taylor series of two functions f and g produces a new Taylor series corresponding the function $f + g$, and similarly for subtraction. The Taylor coefficients of the product and quotient of two functions are slightly more complicated to compute. Before deriving these, we summarize the rules for Taylor arithmetic:

$$\begin{aligned}
(f + g)_k &= f_k + g_k \\
(f - g)_k &= f_k - g_k \\
(f \times g)_k &= \sum_{i=0}^{k} f_i g_{k-i} \\
(f \div g)_k &= \frac{1}{g_0}\left(f_k - \sum_{i=0}^{k-1} (f \div g)_i g_{k-i}\right).
\end{aligned} \tag{4.7}$$

Note that, when dividing, we must[2] have $g_0 \neq 0$, which corresponds to the restriction $v \neq 0$ in (4.6) and (4.2).

To prove the rule for multiplication, we simply write

$$\sum_{k=0}^{\infty} f_k(x - x_0)^k \sum_{k=0}^{\infty} g_k(x - x_0)^k = \sum_{k=0}^{\infty} (f \times g)_k (x - x_0)^k$$

from which the coefficient $(f \times g)_k$ is obtained by gathering all powers $(x - x_0)^k$ from the left-hand side:

$$\sum_{i=0}^{k} f_i(x - x_0)^i g_{k-i}(x - x_0)^{k-i} = (f \times g)_k (x - x_0)^k.$$

The rule for division is obtained in a similar fashion: by definition, we have

$$\sum_{k=0}^{\infty} f_k(x - x_0)^k / \sum_{k=0}^{\infty} g_k(x - x_0)^k = \sum_{k=0}^{\infty} (f \div g)_k (x - x_0)^k.$$

[1] For a complete `C++` implementation of second-order automatic differentiation, see [CXSC].

[2] Actually, if $g_i = f_i = 0$ for $i = 0, \ldots, m - 1$ and $g_m \neq 0$, then the division can be carried out according to l'Hopital's rule.

Multiplying both sides with the Taylor series for g produces

$$\sum_{k=0}^{\infty} f_k(x-x_0)^k = \sum_{k=0}^{\infty}(f \div g)_k(x-x_0)^k \sum_{k=0}^{\infty} g_k(x-x_0)^k,$$

and by the rule for multiplication, we have

$$f_k = \sum_{i=0}^{k}(f \div g)_i g_{k-i} = \sum_{i=0}^{k-1}(f \div g)_i g_{k-i} + (f \div g)_k g_0.$$

Solving for $(f \div g)_k$ produces the desired result.

In order to be able to compute with this new arithmetic, we need to know how constants and the independent variable x are treated. Seen as functions, these have particularly simple Taylor expansions:

$$\begin{aligned} x &= x_0 + 1\cdot(x-x_0) + 0\cdot(x-x_0)^2 + \cdots + 0\cdot(x-x_0)^k + \ldots, \\ c &= c \ + 0\cdot(x-x_0) + 0\cdot(x-x_0)^2 + \cdots + 0\cdot(x-x_0)^k + \ldots. \end{aligned} \tag{4.8}$$

Exercise 4.2.1 *Write a program module that implements the data type* `taylor` *defined by the rules (4.7) and (4.8). Use the module to compute high-order derivatives of the rational function appearing in Example 4.1.1 for several expansion points.*

Exercise 4.2.2 *Extend the rule for division of (4.7), incorporating the case $g_i = f_i = 0$ for $i = 0, \ldots, m-1$ and $g_m \neq 0$. Explain how this extension corresponds to l'Hopital's rule.*

Exercise 4.2.3 *Write down the formal expression for $f * f$ using the rule for multiplication of (4.7). Using the appearing symmetry, find a more efficient formula for computing the square f^2 of a function f.*

4.2.1 Derivatives of standard functions

Extending these ideas to the standard functions is not difficult. Let us start with exponentiation. Given a function g whose Taylor series is known, how do we compute the Taylor series for e^g? Let us formally write

$$g(x) = \sum_{k=0}^{\infty} g_k(x-x_0)^k \quad \text{and} \quad e^{g(x)} = \sum_{k=0}^{\infty}(e^g)_k(x-x_0)^k,$$

and use the fact that

$$\frac{d}{dx}e^{g(x)} = g'(x)e^{g(x)}. \tag{4.9}$$

Plugging the formal expressions for $g'(x)$ and $e^{g(x)}$ into (4.9) produces

$$\sum_{k=1}^{\infty} k(e^g)_k(x-x_0)^{k-1} = \sum_{k=1}^{\infty} kg_k(x-x_0)^{k-1} \sum_{k=0}^{\infty} (e^g)_k(x-x_0)^k,$$

which, after multiplying both sides with $(x-x_0)$, becomes

$$\sum_{k=1}^{\infty} k(e^g)_k(x-x_0)^k = \sum_{k=1}^{\infty} kg_k(x-x_0)^k \sum_{k=0}^{\infty} (e^g)_k(x-x_0)^k.$$

Using the rule for products from (4.7) then yields

$$k(e^g)_k = \sum_{i=1}^{k} ig_i(e^g)_{k-i} \qquad (k>0)$$

Since we know that the constant term is given by $(e^g)_0 = e^{g_0}$, we arrive at:

$$(e^g)_k = \begin{cases} e^{g_0} & \text{if } k=0, \\ \frac{1}{k}\sum_{i=1}^{k} ig_i(e^g)_{k-i} & \text{if } k>0. \end{cases} \tag{4.10}$$

The natural logarithm of a function $\ln g(x)$ can be obtained in a similar fashion. Using the relation

$$\frac{d}{dx}\ln g(x) = \frac{g'(x)}{g(x)},$$

we find that

$$\sum_{k=1}^{\infty} k(\ln g)_k(x-x_0)^{k-1} = \sum_{k=1}^{\infty} kg_k(x-x_0)^{k-1} / \sum_{k=0}^{\infty} g_k(x-x_0)^k.$$

Multiplying both sides with $(x-x_0)$, and rearranging produces

$$\sum_{k=1}^{\infty} k(\ln g)_k(x-x_0)^k \sum_{k=0}^{\infty} g_k(x-x_0)^k = \sum_{k=1}^{\infty} kg_k(x-x_0)^k.$$

Using the rule for multiplication from (4.7), we have

$$\sum_{i=1}^{k} i(\ln g)_i g_{k-i} = kg_k,$$

and solving for $(\ln g)_k$ yields[3]

$$(\ln g)_k = \frac{1}{g_0}\left(g_k - \frac{1}{k}\sum_{i=1}^{k-1} i(\ln g)_i g_{k-i}\right) \qquad (k>0).$$

[3]When $k=1$, the sum appearing in the formula for $(\ln g)_k$ is empty, and evaluates to zero.

Since we know that the constant term is given by $(\ln g)_0 = \ln g_0$, we arrive at:

$$(\ln g)_k = \begin{cases} \ln g_0 & \text{if } k = 0, \\ \frac{1}{g_0}\left(g_k - \frac{1}{k}\sum_{i=1}^{k-1} i(\ln g)_i g_{k-i}\right) & \text{if } k > 0. \end{cases} \tag{4.11}$$

We can now define exponentiation through the equality

$$g(x)^{f(x)} = e^{f(x)\ln g(x)}. \tag{4.12}$$

If, however, the exponent is a constant, $f(x) = a$, we can find a more effective means for computing $g(x)^a$. We will use the fact that we have

$$\frac{d}{dx}g(x)^a = ag'(x)g(x)^{a-1} \quad \text{or} \quad g(x)\frac{d}{dx}g(x)^a = ag'(x)g(x)^a. \tag{4.13}$$

The Taylor series representation for the second part of (4.13) is

$$\sum_{k=0}^{\infty} g_k(x-x_0)^k \sum_{k=1}^{\infty} k(g^a)_k(x-x_0)^{k-1} = a\sum_{k=1}^{\infty} kg_k(x-x_0)^{k-1}\sum_{k=0}^{\infty}(g^a)_k(x-x_0)^k.$$

Multiplying both sides with $(x-x_0)$, and using the rule for multiplication from (4.7) gives

$$\sum_{i=0}^{k} g_i(k-i)(g^a)_{k-i} = a\sum_{i=1}^{k} ig_i(g^a)_{k-i}$$

Extracting the first term of the left-hand side produces

$$g_0k(g^a)_k = \sum_{i=1}^{k}\big(aig_i(g^a)_{k-i} - g_i(k-i)(g^a)_{k-i}\big) = \sum_{i=1}^{k}\big((a+1)i-k\big)g_i(g^a)_{k-i}.$$

Summarizing, we arrive at:

$$(g^a)_k = \begin{cases} g_0^a & \text{if } k = 0, \\ \frac{1}{g_0}\sum_{i=1}^{k}\big(\frac{(a+1)i}{k} - 1\big)g_i(g^a)_{k-i} & \text{if } k > 0. \end{cases} \tag{4.14}$$

Unfortunately, neither formula (4.12) nor (4.14) is suitable for the situation when $g_0 = 0$ and $a > 0$, despite the fact that $g(x)^a$ may be well-defined.

The Taylor coefficients of $\sin g(x)$ and $\cos g(x)$ must be computed in parallel. Following the usual procedure of differentiating, and multiplying both sides by $(x-x_0)$, we end up with the formulas:

$$(\sin g)_k = \begin{cases} \sin g_0 & \text{if } k = 0, \\ \frac{1}{k}\sum_{i=1}^{k} ig_i(\cos g)_{k-i} & \text{if } k > 0. \end{cases}$$

$$(\cos g)_k = \begin{cases} \cos g_0 & \text{if } k = 0, \\ -\frac{1}{k}\sum_{i=1}^{k} ig_i(\sin g)_{k-i} & \text{if } k > 0. \end{cases} \tag{4.15}$$

By using the rule for division from (4.7), it is straight-forward to obtain the Taylor coefficients of $\tan g(x)$.

$$(\tan g)_k = \begin{cases} \tan g_0 & \text{if } k = 0, \\ \frac{1}{\cos^2 g_0} \left(g_k - \frac{1}{k} \sum_{i=1}^{k-1} i (\tan g)_i (\cos^2 g)_{k-i} \right) & \text{if } k > 0. \end{cases} \tag{4.16}$$

For the inverse trigonometric functions, we use well-known the rules of differentiation

$$\begin{aligned} \frac{d}{dx} \arcsin g(x) &= \frac{g'(x)}{\sqrt{1 - (g(x))^2}} \\ \frac{d}{dx} \arccos g(x) &= \frac{-g'(x)}{\sqrt{1 - (g(x))^2}} \\ \frac{d}{dx} \arctan g(x) &= \frac{g'(x)}{1 + (g(x))^2} \end{aligned}$$

which yield the recursive formulas:

$$(\arcsin g)_k = \begin{cases} \arcsin g_0 & \text{if } k = 0, \\ \frac{1}{\sqrt{1-(g_0)^2}} \left(g_k - \frac{1}{k} \sum_{i=1}^{k-1} i (\arcsin g)_i (\sqrt{1 - g^2})_{k-i} \right) & \text{if } k > 0, \end{cases}$$

$$(\arccos g)_k = \begin{cases} \arccos g_0 & \text{if } k = 0, \\ \frac{-1}{\sqrt{1-(g_0)^2}} \left(g_k + \frac{1}{k} \sum_{i=1}^{k-1} i (\arccos g)_i (\sqrt{1 - g^2})_{k-i} \right) & \text{if } k > 0, \end{cases}$$

$$(\arctan g)_k = \begin{cases} \arctan g_0 & \text{if } k = 0, \\ \frac{1}{1+(g_0)^2} \left(g_k - \frac{1}{k} \sum_{i=1}^{k-1} i (\arctan g)_i (1 + g^2)_{k-i} \right) & \text{if } k > 0. \end{cases}$$

Exercise 4.2.4 *Extend your implementation of the data type* `taylor` *to incorporate the recursive rules for the standard functions presented above. Use the module to compute the quantity* $f^{(4)}(1)$*, where* $f(x) = (5 + \cos^2 3x)^{e^x + \sin 7x}$.

[Answer: -20805870.26519189]

4.3 Higher order enclosures

If f is n times continuously differentiable on a domain $[x]$, then we can expand f in its Taylor series around any point $x_0 \in [x]$:

$$\begin{aligned} f(x) &= f(x_0) + f'(x_0)(x - x_0) + \cdots + \frac{f^{(n-1)}(x_0)}{(n-1)!}(x - x_0)^{n-1} + \frac{f^{(n)}(\zeta)}{n!}(x - x_0)^n \\ &= f_0 + f_1(x - x_0) + \cdots + f_{n-1}(x - x_0)^{n-1} + \frac{f^{(n)}(\zeta)}{n!}(x - x_0)^n \end{aligned} \tag{4.17}$$

where $x \in [x]$ and ζ is somewhere between x and x_0. By combining automatic differentiation and interval arithmetic, we obtain enclosures of the Taylor coefficients, which we denote $F_0, F_1, \ldots, F_n$. In particular, we have $\frac{f^{(n)}(\zeta)}{n!} \in F_n$ for all $\zeta \in [x]$. Therefore we can bound the remainder term in (4.17) to obtain the enclosure

$$f(x) \subseteq f_0 + f_1(x - x_0) + \cdots + f_{n-1}(x - x_0)^{n-1} + F_n([x] - x_0)^n, \tag{4.18}$$

valid for all $x_0, x \in [x]$.

Note that, if we select $x_0 = \mathrm{mid}([x])$, and let $r = \mathrm{rad}([x])$, then the remainder term is enclosed by the interval $F_n r^n[-1, 1]$ if n is odd, and $F_n r^n[0, 1]$ if n is even. In both cases, the interval widths scale like r^n.

To emphasize the splitting of f into a real-valued polynomial part and an interval-valued remainder term, we introduce the special notations

$$P^{n-1}_{f,x_0}(h) \stackrel{\text{def}}{=} f_0 + f_1 h + \cdots + f_{n-1}h^{n-1},$$
$$I^n_{f,[x]}(h) \stackrel{\text{def}}{=} F^{(n)}([x])[-h, h]^n.$$

We then have the following inclusion:

$$f(x_0 + h) \in P^{n-1}_{f,x_0}(h) + I^n_{f,[x]}(h), \tag{4.19}$$

which also provides an enclosure of the range of f over $[x] = x_0 + [-r, r]$:

$$R(f; [x]) \subseteq R\big(P^{n-1}_{f,x_0}; [-r, r]\big) + I^n_{f,[x]}(r).$$

The idea of separating a function into a polynomial approximation and an interval enclosure of the remainder term, as in (4.19), is really not new: we have already seen this in the centered forms, where the polynomial part has degree zero[4].

[4] A more structured development of so called Taylor models was introduced by Berz and Makino, see e.g. [Be97] and [BM98]. They take the whole concept further by defining a Taylor-model arithmetic, which allows them to compute with objects of the form similar to (4.19) as intrinsic quantities. Pursuing this interesting direction of research would, however, take us too far afield from our present path.

4.4 Computer Lab III

Problem 1. Implement the first-order automatic differentiation rules (4.2), (4.3), and (4.5) from the lecture notes. The underlying number field should be of type `double`. Use your implementation to compute the couple $(f(x_0), f'(x_0))$, where $x_0 = 1$, and f is given in Problem 2 of Computer Lab II.

Problem 2. Repeat Problem 1, but this time the underlying number field should be of type `interval`. Now use your implementation to compute the eleven couples $(f([x_k]), f'([x_k]))$ for $k = 0, \ldots, 10$, where $[x_k] = 1 + 2^{-k}[-1, 1]$.

Problem 3. Write a small set of routines supporting taylor series arithmetic, as defined by equations (4.7) and (4.8). The underlying number field should be of type `double`. Use your implementation to compute $g^{(k)}(1)$ for $k = 0, \ldots, 5$, where

$$g(x) = \frac{7x - (x+1)^2}{3x - 2}.$$

[Recall that $g^{(k)}(x) = \frac{d^k g}{dx^k}(x)$.]

Problem 4. Using the formulas from Section 4.2.1, extend your work from Problem 3 by implementing taylor series arithmetic versions of the standard functions e^x, $\log x$, x^n $(n \in \mathbb{Z})$, x^α $(\alpha \in \mathbb{R} \setminus \mathbb{Z})$, $\sin x$, $\cos x$, and $\tan x$. Finally, compute $f^{(k)}(1)$ for $k = 0, \ldots, 5$ where f is given in Problem 2 of Computer Lab II.

Chapter 5

Interval analysis in action

In this chapter, we will illustrate how interval analysis can be successfully applied to various areas of mathematics.

5.1 Root finding

We will start by presenting methods for locating all roots of a function f over a given domain $[x]$. If the function is merely continuous, a bisection-type method is usually a good choice, whereas if f is differentiable, a Newton-type method may be preferable. We will begin by describing the simpler bisection-type method.

5.1.1 Divide and conquer

Let $f\colon I \to \mathbb{R}$ be a continuous, elementary function. By Theorem 3.1.11 we know that, if $[x] \subseteq I$ and $F([x])$ exists, then $R(f;[x]) \subseteq F([x])$. As pointed out earlier, this is equivalent to the statement "$y \notin F([x]) \Rightarrow y \notin R(f;[x])$". Our approach to finding all roots of f will be an adaptive bisection scheme, where subsets of the initial domain are either proved *not* to contain any roots, whereby they are discarded from the search, or are bisected and kept for further study. When all remaining subintervals are smaller then some pre-defined tolerance, the search is terminated. We are then left with a (possibly empty) collection of intervals whose union covers all possible roots of f within $[x]$. A simple implementation of this search is presented in program (5.1.1).

Upon the termination of the program (5.1.1), all elements of the output stream will be possible candidates for root enclosures for f. That is, an interval printed from the program *may* contain a root of f. What is *absolutely sure* is that no root of f can exist outside the union of the printed elements.

```
#include <iostream>
#include "Interval.h"
#include "Functions.h"
using namespace std;
typedef INTERVAL (*pfcn)(INTERVAL);

void bisect(pfcn f, INTERVAL X, double Tol) {
  if ( 0.0 <= f(X) )
    if ( Diam(X) < Tol )
      cout « X « endl;
    else {
      bisect(f, INTERVAL(Inf(X), Mid(X)), Tol);
      bisect(f, INTERVAL(Mid(X), Sup(X)), Tol);
    }
}

INTERVAL function(INTERVAL x) {
  return sin(x)*(x - cos(x));
}

int main(int argc, char * argv[])
{
  INTERVAL   X(atof(argv[1]), atof(argv[2]));
  double   Tol(atof(argv[3]));

  bisect(function, X, Tol);
  return 0;
}
```

Figure 5.1: An implementation of a recursive interval bisection scheme.

Example 5.1.1 *Consider the function* $f(x) = \sin x(x - \cos x)$ *over the domain* $[-10, 10]$*, illustrated in Figure 5.2(a). The function clearly has eight roots in the domain:* $\{\pm 3\pi, \pm 2\pi, \pm\pi, 0, x^*\}$*, where* x^* *is the unique (positive) root of* $x - \cos x = 0$*. Executing program (5.1.1) with tolerance* 0.001 *produces the nine intervals listed below. Note, however, that intervals 4 and 5 are adjacent. This always happens when a root is located exactly at a bisection point of the domain.*

```
Domain        : [-10,10]
Tolerance     : 0.001
Function calls: 227
Root list     :
  1: [-9.42505,-9.42444]  4: [-0.00061,+0.00000]  7: [+3.14148,+3.14209]
  2: [-6.28357,-6.28296]  5: [+0.00000,+0.00061]  8: [+6.28296,+6.28357]
  3: [-3.14209,-3.14148]  6: [+0.73853,+0.73914]  9: [+9.42444,+9.42505]
```

Exercise 5.1.2 *Write your own bisection routines: one using the interval tech-*

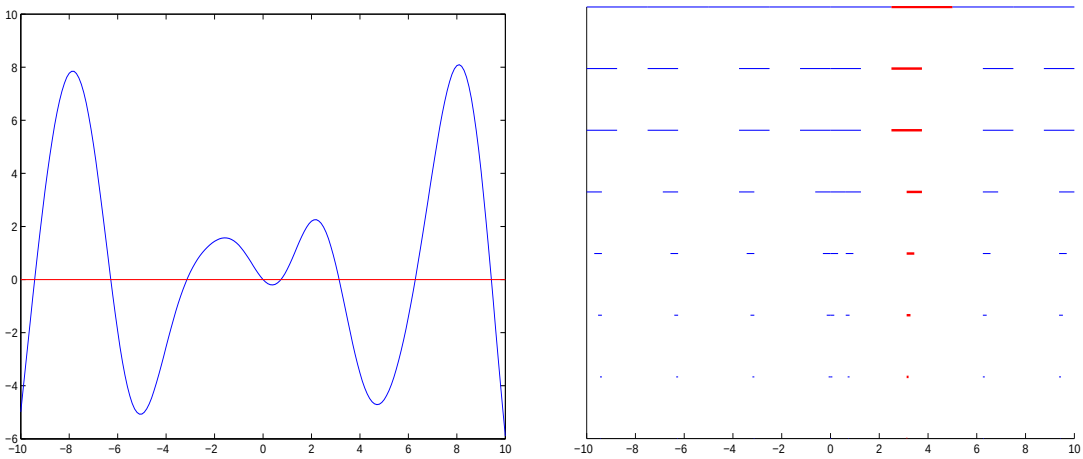


Figure 5.2: (a) The function $f(x) = \sin x(x - \cos x)$ over the domain $[-10, 10]$; (b) Increasingly tight root enclosures.

niques described in this section, and one using the standard floating point approach. Evaluate the two routines by comparing the number of function evaluations needed to locate all roots within some tolerance. The article [Co77] provides some interesting ideas.

Note that if we use a centered form when computing the enclosure of $R(f; [\tilde{x}])$, we can use the information provided from the derivative enclosure $F'([\tilde{x}])$. Recall that, if $0 \notin F'([\tilde{x}])$, then f is monotone over $[\tilde{x}]$. If this is the case, then f either has a unique root in $[\tilde{x}]$, or f has no roots in $[\tilde{x}]$ at all. The former happens when f has opposite signs at the endpoints of $[\tilde{x}]$; the latter when the signs are equal.

Exercise 5.1.3 *Rewrite program (5.1.1), incorporating checks for monotonicity.*

5.1.2 Newton's method

Assuming now that the function is differentiable, we will continue to illustrate the power of interval arithmetic in root finding. The basic tool for finding roots of non-linear equations is Newton's method, which we will begin by describing in the real-valued setting.

Let $f\colon [x] \to \mathbb{R}$ be a continuously differentiable function, and suppose that $x^* \in [x]$ is a root of f, i.e., $f(x^*) = 0$. Given an initial guess $x_0 \in [x]$, our strategy is to compute the intersection between the tangent at $f(x_0)$ and the x-axis. The tangent is given by the linear equation

$$t(x) = f(x_0) + f'(x_0)(x - x_0),$$

so the point of intersection is easily computed as

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

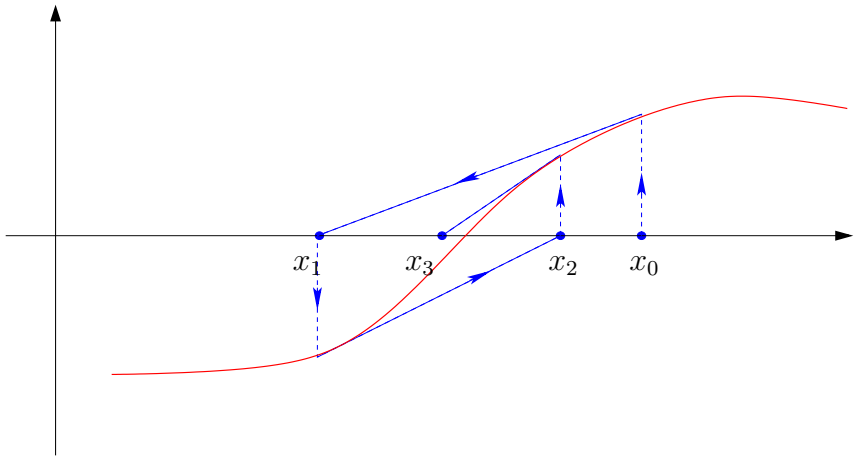


Figure 5.3: The first few iterates of Newton's method.

This expression is well-defined, provided that $f'(x_0) \neq 0$. Repeating this process gives the following sequence (see Figure 5.3):

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \qquad k = 0, 1, 2, \ldots.$$

Theorem 5.1.4 *Assume that $f\colon [x] \to \mathbb{R}$ is twice continuously differentiable with $f'(x) \neq 0$ for all $x \in [x]$, and that f has a unique, simple root x^* within $[x]$. Then, if x_0 is chosen sufficiently close to x^*, the sequence $\{x_k\}_{k=0}^{\infty}$ converges quadratically fast toward x^*, i.e., there exists a constant C such that*

$$\lim_{k\to\infty} x_k = x^* \qquad \textit{and} \qquad |x_{k+1} - x^*| \leq C|x_k - x^*|^2.$$

Proof: Let ε_k denote the error in the iterate x_k, i.e., $\varepsilon_k = x_k - x^*$. Expanding f in a Taylor series centered at x_k gives

$$0 = f(x^*) = f(x_k) + f'(x_k)(x^* - x_k) + \tfrac{1}{2}f''(\zeta)(x^* - x_k)^2,$$

where ζ is between x^* and x_k. Dividing by $f'(x_k)$ then gives

$$\frac{f(x_k)}{f'(x_k)} + x^* - x_k = x^* - x_{k+1} = \frac{-\frac{1}{2}f''(\zeta)(x^* - x_k)^2}{f'(x_k)},$$

which means that

$$|\varepsilon_{k+1}| = \left|\frac{f''(\zeta)}{2f'(x_k)}\right| \varepsilon_k^2 \leq \sup\left\{\left|\frac{f''(x)}{2f'(y)}\right| : x, y \in [x]\right\} \varepsilon_k^2 = C\varepsilon_k^2.$$

We can rewrite the inequality $|\varepsilon_{k+1}| \leq C\varepsilon_k^2$ as $|C\varepsilon_{k+1}| \leq (C\varepsilon_k)^2$, so if $|C\varepsilon_0| < 1$ and $[x^* - \varepsilon_0, x^* + \varepsilon_0] \subseteq [x]$, then it follows by induction that $x_k \in [x]$ for all k, and that

$$|\varepsilon_k| \leq \frac{1}{C}(C\varepsilon_0)^{2^k}.$$

Therefore, if we choose x_0 close enough to x^* to ensure that $|C\varepsilon_0| = C|x_0 - x^*| < 1$, then it follows that x_k tends to x^* quadratically fast. □

It should be pointed out that, unless the assumptions of Theorem 5.1.4 are fulfilled, Newton's method may fail to converge. The iterates can form a periodic orbit, or wander around in an apparently aimless pattern. This should not be viewed as a rare, degenerate situation, but rather as a naturally occurring phenomenon associated to deep mathematical properties of Newton's method[1].

5.1.3 The interval Newton method

Unlike the real-valued Newton method, the interval version always displays very regular behaviour. To begin with, we will assume that $f\colon [x] \to \mathbb{R}$ is a continuously differentiable function, and that $x^* \in [x]$ is a root of f. We will also assume that an interval extension of f' exists and satisfies $0 \notin F'([x])$. In particular, this implies that $f'(x) \neq 0$ throughout $[x]$. Then, for any $x \in [x]$, an application of the Mean Value Theorem gives

$$f(x) = f(x^*) + f'(\zeta)(x - x^*)$$

for some ζ between x and x^*. Since $f(x^*) = 0$ and $f'(\zeta) \neq 0$, we can solve for the root x^*:

$$x^* = x - \frac{f(x)}{f'(\zeta)} \in x - \frac{f(x)}{F'([x])} \stackrel{\text{def}}{=} N([x]; x).$$

Since we are assuming that $x^* \in [x]$, we also have $x^* \in N([x]; x) \cap [x]$ for all $x \in [x]$. The enclosure corresponding to the selection $x = m = \operatorname{mid}([x])$ is called the *interval Newton operator*:

$$N([x]) \stackrel{\text{def}}{=} N([x], m) = m - \frac{f(m)}{F'([x])}.$$

Taking $[x_0] = [x]$ as our initial enclosure of x^*, we define the sequence of intervals

$$[x_{k+1}] = N([x_k]) \cap [x_k], \quad k = 0, 1, 2, \ldots. \tag{5.1}$$

In Figure 5.4 we illustrate the partial construction of this sequence. Note that, by construction, if $x^* \in [x_0]$ then $x^* \in [x_k]$ for all $k \in \mathbb{N}$. This means that we never lose track of the root. The sequence of intervals defined by (5.1) has several other nice properties.

Theorem 5.1.5 (Interval Newton Method) *Assume that $N([x_0])$ is well-defined. If $[x_0]$ contains a root x^* of f, then so do all iterates $[x_k]$, $k \in \mathbb{N}$. Furthermore, the intervals $[x_k]$ form a nested sequence converging to x^*.*

[1] The use of Newton's method to solve cubic polynomials (over $\mathbb{C}$) actually led to the discovery of a simple, deterministic system that exhibits chaotic behaviour.

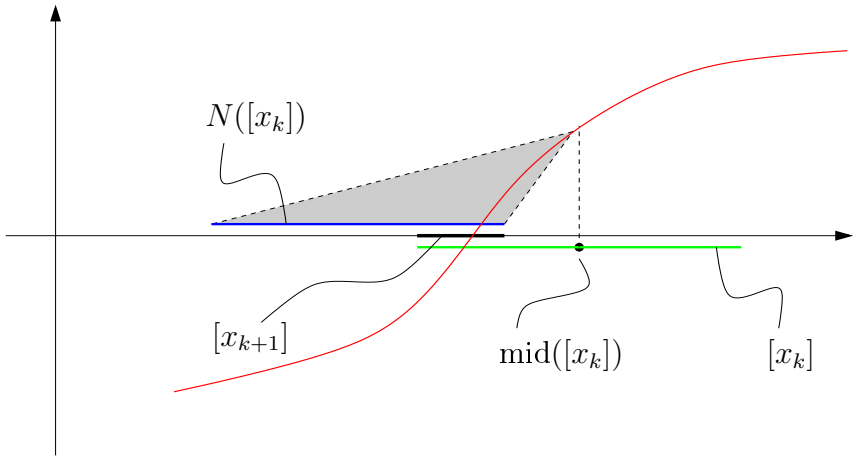


Figure 5.4: One iterate of the interval Newton method.

Proof: As mentioned above, the implication $x^* \in [x_0] \Rightarrow x^* \in [x_k]$, for all $k \in \mathbb{N}$ follows immediately by induction. Also, by (5.1), it is clear that the intervals $[x_k]$ form a nested sequence. We need therefore only prove that the intervals $[x_k]$ converge to x^*. This can happen in two ways: the first scenario is that, at some stage, there is a thin iterate $[x_k] = [x^*, x^*]$, in which case we have $\mathrm{rad}([x_k]) = 0$ and $[x_k] = [x_{k+i}]$ for all $i \in \mathbb{N}$. This happens exactly when $x^* = \mathrm{mid}([x_{k-1}])$. Let us now consider the second scenario, i.e., we assume that $x^* \neq \mathrm{mid}([x_k])$ for all $k \in \mathbb{N}$. The fact that $N([x_0])$ is well-defined implies that $0 \notin F'([x_0])$. Therefore, since $[x_k] \subseteq [x_0]$, it follows that $0 \notin F'([x_k])$ for all $k \in \mathbb{N}$. This implies that the correction term (which now is an interval)

$$\frac{f(\mathrm{mid}([x_k]))}{F'([x_k])}$$

consists entirely of elements of the same sign. As a consequence, the midpoint of $[x_k]$ is not contained in $[x_{k+1}]$, see Figure 5.4. This means that $\mathrm{rad}([x_{k+1}]) < \frac{1}{2}\mathrm{rad}([x_k])$, and the convergence is proved. □

Thus, the sequence (5.1) converges to x^* at least at a linear rate. Under conditions similar to those of Theorem 5.1.4, it is possible to show that the convergence rate is quadratic, i.e., there exists a constant C such that

$$d\big([x_{k+1}], x^*\big) \leq C d\big([x_k], x^*\big)^2.$$

A proof of this statement is given in [Mo66].

One of the most useful properties of the interval Newton operator N is that we are provided with a means of detecting when a region does *not* contain a root of f. As this is a common situation, it is important that we can quickly discard a set on the grounds of it containing no roots. Another important contribution from the properties of N is a simple, verifiable condition that guarantees the existence of a unique root within an interval.

Theorem 5.1.6 *Let $f \in C^2([x], \mathbb{R})$, and assume that $N([x])$ is well-defined for some $[x] \in \mathbb{IR}$. Then the following statements hold:*

(1) if $N([x]) \cap [x] = \emptyset$, then $[x]$ contains no roots of f;

(2) if $N([x]) \subseteq [x]$, then $[x]$ contains exactly one root of f.

Proof: We begin by proving part (1). This statement follows as a consequence of Theorem 5.1.5, since if $[x]$ contains a root x^*, then so does $N([x])$, which means that $x^* \in N([x]) \cap [x]$. Therefore, if $N([x]) \cap [x] = \emptyset$, then $[x]$ cannot contain a root of f.

To prove part (2), we first recall that $N([x])$ being well-defined implies that f is monotone on $[x]$. Therefore, since f is continuous on $[x]$, there can be *at most* one root in $[x]$, so we only have to establish the existence of a root $x^* \in [x]$. The inclusion $N([x]) \subseteq [x]$ translates to $m - \frac{f(m)}{F'([x])} \subseteq [x]$, which implies that

$$m - \frac{f(m)}{f'(\zeta)} \subseteq [x] \quad \text{for all} \quad \zeta \in [x].$$

But $m - \frac{f(m)}{f'(\zeta)}$ is the solution to

$$t_\zeta(x) = f(m) + f'(\zeta)(x - m) = 0.$$

Recall that $t_\zeta(x)$ describes the line passing through $(m, f(m))$ and having slope $f'(\zeta)$. Since we are assuming that f' is continuous on the compact domain $[x]$, there are points ζ^+ and ζ^- in $[x]$ where the derivative attains is maximum and minimum, respectively. Thus, for all $x \in [x]$, the function value $f(x)$ lies between $t_{\zeta^+}(x)$ and $t_{\zeta^-}(x)$, see Figure 5.4. Therefore, since $f'(\zeta) \in F'([x])$, the graph of f must intersect the x-axis somewhere within the set

$$Z([x]) \stackrel{\text{def}}{=} \{t_\zeta(x) = 0 \colon \zeta \in [x]\},$$

i.e., $x^* \in Z_\zeta$. But, by construction, $N([x])$ contains $Z([x])$. Hence $x^* \in N([x]) \subseteq [x]$, as claimed. □

Example 5.1.7 *Consider the polynomial*

$$f(x) = -2.001 + 3x - x^3,$$

which has derivative $f'(x) = 3(1 - x^2)$. If we choose $[x_0] = [-3, -3/2]$, then $F'([x_0]) = [-24, -15/4]$, so $N([x_0])$ is well-defined, and Theorem 5.1.5 holds.

The following output was generated by a short program implementing the interval Newton method. Of course, all interval operations are performed with directed rounding.

```
X(0) = [-3.000000000000000,-1.500000000000000]; rad = 7.50000e-01
X(1) = [-2.140015625000001,-1.546099999999996]; rad = 2.96958e-01
X(2) = [-2.140015625000001,-1.961277398284108]; rad = 8.93691e-02
X(3) = [-2.006849239640351,-1.995570580247208]; rad = 5.63933e-03
X(4) = [-2.000120104486270,-2.000103608530276]; rad = 8.24798e-06
X(5) = [-2.000111102890393,-2.000111102873815]; rad = 8.28893e-12
X(6) = [-2.000111102881727,-2.000111102881724]; rad = 1.55431e-15
X(7) = [-2.000111102881727,-2.000111102881724]; rad = 1.55431e-15
Finite convergence!
Unique root guaranteed within the enclosure: -2.00011110288172 +- 1.555e-15
```

There are two things to notice here. First of all, the intervals `X(6)` *and* `X(7)` *are identical. This is due to the finiteness of the set of floating points utilized in the computations. Here is a rare instance where the finiteness comes in handy: it provides us with a good stopping condition.*

Secondly, by Theorem 5.1.5, it follows that `X(7)` *contains the unique root of f in the domain $[-3, -3/2]$. This result would require some theoretical work to prove in a traditional manner.*

To see how our program handles the case when there are no roots, we pick a different initial region.

Example 5.1.8 *Using the same function as in the previous example, we take $[x_0] = [3/2, 5/2]$. Since $F'([x_0]) = [-57/4, -15/4]$ does not contain zero, Theorem 5.1.5 still holds. We now receive the following output:*

```
X(0) = [+1.500000000000000,+2.500000000000000]; rad = 5.00000e-01
X(1) = [+1.500000000000000,+1.745968253968255]; rad = 1.22984e-01
Empty intersection. No roots in the domain.
```

Thus, according to Corollary 5.1.6, there are no roots in the region $[3/2, 5/2]$.

If, however, we would attempt to find a root by the real-valued Newton's method with $x_0 = 2 \in [3/2, 5/2]$, the resulting sequence would be

```
Enter the initial value: 2
X(1) = 1.5554444444444444
X(2) = 1.2976090310434865
X(3) = 1.1547419130568084
X(4) = 1.0782233643492227
X(5) = 1.037551943368803
...
X(28) = 1.0089842691534474
X(29) = 0.98603084839013477
X(30) = 1.0050467709456032
```

This sequence never settles down. Instead it displays erratic behaviour! As we already have pointed out, this is no anomaly, but an inherent property of the real-valued Newton method.

Exercise 5.1.9 *Write your own interval Newton routine. To handle more general situations, use a bisection scheme with monotonicity checks to single out subdomains where* $0 \notin F'([x])$.

5.1.4 The extended interval Newton method

Using the extended interval division, as described in Section 2.3.4, we can compute

$$N([x]) = \mathrm{mid}(x) - \frac{f(\mathrm{mid}(x))}{F'([x])}.$$

even when $0 \in F'([x])$. This situation occurs, e.g., when the domain $[x]$ contains several roots of f. In a situation where all roots of a function are sought, it is extremely convenient to be able to handle this type of "division by zero" in a seamless manner.

At any stage of the iteration, the extended interval Newton operator may produce one or two unbounded intervals, depending on the exact position of the interval $F'([x_k])$. After intersecting with the current interval $[x_k]$, the result is therefore one or two compact intervals, i.e., we either have

$$N([x_k]) \cap [x_k] = [x_{k+1}]$$

as usual, or

$$N([x_k]) \cap [x_k] = [x_{k+1}^{(l)}] \cup [x_{k+1}^{(r)}],$$

as illustrated in Figure 5.5. In some (very rare) situations we may have $N([x_k]) = [x_k]$, in which case we simply bisect $[x_k]$ before continuing. The great advantage with this extension is that we now can find *all* roots of a function, within a given domain $[x]$. We illustrate this in the following example.

Example 5.1.10 *Consider the function* $f(x) = \sin x(x - \cos x)$ *used in Example 5.1.1. Again, we will consider the domain* $[x] = [-10, 10]$. *Recall that the function has eight roots in* $[x]$: $\{\pm 3\pi, \pm 2\pi, \pm\pi, 0, x^*\}$, *where* x^* *is the unique (positive) root of* $x - \cos x = 0$. *Running an extended interval Newton search with tolerance* 2^{-10} *produces the nine intervals listed below. Note, however, that intervals 5 and 9 are adjacent. As remarked earlier, this always happens when a root is located exactly at a bisection point of the domain.*

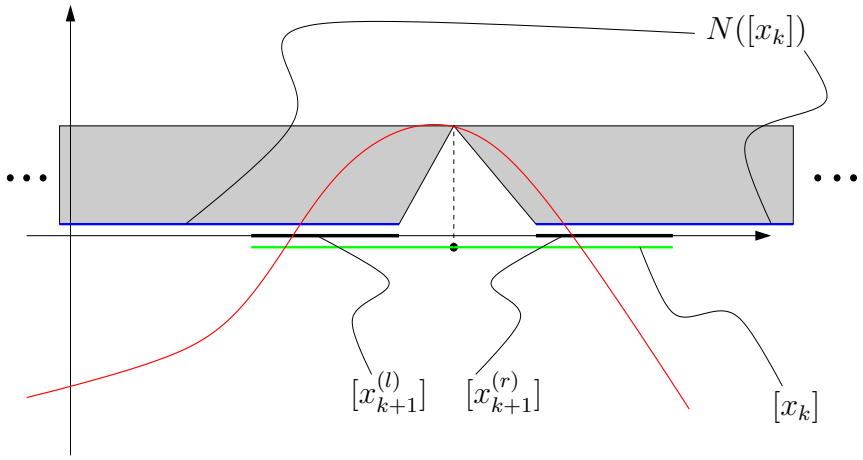


Figure 5.5: One iterate of the extended interval Newton method.

```
Domain               : [-10, +10]
Tolerance            : 9.765625e-04  (2^-10)
Function calls       : 39
 Unique root in the interval [-6.28395516124695064,-6.28268172307243855]
 Unique root in the interval [-9.42494303272712486,-9.42466416085800240]
 Unique root in the interval [+9.42469385546710114,+9.42492302076889210]
 Unique root in the interval [+6.28318404553732801,+6.28318680558727749]
Maybe a root in the interval [-0.00022342911344772,+0.00000000000000000]
 Unique root in the interval [-3.14160246753701244,-3.14158303523038152]
 Unique root in the interval [+3.14159233975917740,+3.14159301982135242]
 Unique root in the interval [+0.73905272198116201,+0.73913804091868240]
Maybe a root in the interval [+0.00000000000000000,+0.00009861499990579]
```

If we change the domain ever so slightly to $[x] = [-10, 10.001]$*, we get a confirmation that the function indeed has exactly eight roots in* $[x]$*:*

```
Domain               : [-10, +10.001]
Tolerance            : 9.765625e-04  (2^-10)
Function calls       : 39
 Unique root in the interval [-6.28396275881754285,-6.28267595726734562]
 Unique root in the interval [+9.42469464339922958,+9.42492109675931999]
 Unique root in the interval [+6.28318411250503672,+6.28318673047858933]
 Unique root in the interval [-9.42494326359499546,-9.42466402939086834]
 Unique root in the interval [+3.14159234580190461,+3.14159301300500626]
 Unique root in the interval [+0.73905039495704095,+0.73914183386738308]
 Unique root in the interval [-0.00021800677132035,+0.00008258245398134]
 Unique root in the interval [-3.14160239467736169,-3.14158310518228356]
```

Exercise 5.1.11 *Write your own extended interval Newton routine. This requires that you implement the extended interval division as described in Section 2.3.4.*

5.1.5 The Krawczyk method

An attractive alternative to the interval Newton method is that of *Krawczyk*, which avoids the perils of having to divide by the enclosure $F'([x])$ when computing the Newton iterates. In light of what we have done so far, the idea is very simple: Assuming that $f \in C^1([x], \mathbb{R})$ has a root $x^* \in [x]$, we Taylor expand around x^* to get $f(x) = f(x^*) + f'(\zeta)(x - x^*) = f'(\zeta)(x - x^*)$ for some ζ between x and x^*. Next, we multiply the expansion by some (finite) constant C:

$$Cf(x) = Cf'(\zeta)(x - x^*),$$

and add $x^* - x$ to both sides of the equation:

$$x^* - x + Cf(x) = x^* - x + Cf'(\zeta)(x - x^*).$$

After a rearrangement, this reduces to

$$x^* = x - Cf(x) - \big(1 - Cf'(\zeta)\big)(x - x^*).$$

Now, although we neither know the root x^* nor the point ζ, we do know that both points belong to the domain $[x]$. Thus we can enclose the root via

$$x^* \in x - Cf(x) - \big(1 - CF'([x])\big)(x - [x]) \stackrel{\text{def}}{=} K([x], x, C).$$

We have now proved that any zero $x^* \in [x]$ of f is also enclosed by the Krawczyk operator $K([x], x, C)$ for any $x \in [x]$ and C finite. At this point, the benefit of Krawczyk's formulation should be apparent: we no longer have to divide by $F'([x])$, which (as we have seen) is a potential danger. Good choices for x and C are $x = m = \text{mid}([x])$ and $C = 1/f'(m)$, which results in the operator

$$K([x]) \stackrel{\text{def}}{=} K([x], m, 1/f'(m)) = m - \frac{f(m)}{f'(m)} - \left(1 - \frac{F'([x])}{f'(m)}\right)[-r, r],$$

where we use the notation $r = \text{rad}([x])$.

Given an initial search region $[x_0]$ for a root x^*, we define the sequence of intervals

$$[x_{k+1}] = K([x_k]) \cap [x_k], \quad k = 0, 1, 2, \ldots \tag{5.2}$$

forming the *Krawczyk iterates* of $[x_0]$. As long as $f'(m_k) \neq 0$ for all $k \in \mathbb{N}$, the sequence is well-defined. Analogous to Theorems 5.1.5 and 5.1.6, we have

Theorem 5.1.12 *Assume that $K([x])$ is well-defined. Then the following statements hold:*

(1) if $[x]$ contains a root x^* of f, then so does $K([x]) \cap [x]$;

(2) if $K([x]) \cap [x] = \emptyset$, then $[x]$ contains no roots of f;

(3) if $K([x]) \subseteq \text{int}([x])$*, then* $[x]$ *contains exactly one root of* f*.*

Proof: (1) holds true by construction. (2) follows immediately from the contrapositive statement of (1). To prove (3), we note that $K([x]) \subseteq \text{int}([x])$ implies that the width of the Krawczyk image satisfies $w(K([x])) < w([x]) = w([-r, r])$. On the other hand, we also have

$$w(K([x])) = w\left(\left(1 - \frac{F'([x])}{f'(m)}\right)[-r, r]\right),$$

which implies that $F'([x])/f'(m) \subset (0, 2)$. Since $K([x])$ is well-defined, we know that $f'(m) \neq 0$. It follows that $0 \notin F'([x])$, which means that f' is non-zero on $[x]$, so f can have at most one zero in $[x]$. To see that f indeed has a zero in $[x]$, we note that the Krawczyk operator is simply the midpoint form of the operator $G([x], x) = x - f(x)/f'(\text{mid}([x]))$. Following Section 3.2, we have

$$G_m([x], x) = m - \frac{f(m)}{f'(m)} + (1 - \frac{F([x])}{f'(m)})([x] - m) = K([x]).$$

Therefore, $G([x], x) \in K([x]) \subseteq \text{int}([x])$ for all $x \in [x]$. By the Intermediate Value Theorem, this implies that G (and thus f) has a zero in $[x]$. This completes the proof. □

We end this section with a simple implementation of the Krawczyk method, merged with a bisection scheme. Although this code is not optimal with regards to the number of calls to the Krawczyk operator K, its simple structure has some merit.

We will use the same function $f(x) = \sin x(x - \cos x)$ as in Examples 5.1.1 and 5.1.10. Note that the extended interval Newton method is more efficient than the Krawczyk method, in that it requires fewer function evaluations. In fact, the Newton methods require one derivative evaluation per function evaluation, whereas for the Krawczyk operator the ratio is 2 : 1.

```
Domain          : [-10, 10]
Tolerance       : 9.765625e-04  (2^-10)
Function calls  : 89
 Unique root in the interval [-9.424778380712580,-9.424777714952242]
 Unique root in the interval [-6.283186564304483,-6.283184617341837]
Maybe a root in the interval [-3.141722588593495,-3.141306012318185]
Maybe a root in the interval [-0.000000581733623,+0.000000000000000]
Maybe a root in the interval [+0.000000000000000,+0.000253365746756]
 Unique root in the interval [+0.738912754470971,+0.739439826304897]
 Unique root in the interval [+3.141585088281115,+3.141614165426731]
 Unique root in the interval [+6.283180363246271,+6.283193001393917]
 Unique root in the interval [+9.424777809870829,+9.424778213686277]
```

All eight roots can be securely enclosed if we decrease the tolerance and shift the domain. This only incurs a small increase in the computational cost.

```
#include <iostream>
#include "Interval.h"
#include "Functions.h"
using namespace std;
typedef INTERVAL (*pfcn)(INTERVAL, int);

INTERVAL function(INTERVAL X, int n) {
  if ( n == 0 ) return Sin(X)*(x - Cos(X));
  else          return Sin(X)*(1 + Sin(X)) + Cos(X)*(X - Cos(X));
}

INTERVAL F  (INTERVAL X) { return function(X,0); }
INTERVAL DF (INTERVAL X) { return function(X,1); }

INTERVAL K (pfcn f, INTERVAL X)
{
  INTERVAL   m(Mid(X));
  INTERVAL DFm(DF(m));
  return m - F(m)/DFm + (1 - DF(X)/DFm)*(x - m);
}

void krawczyk(pfcn f, INTERVAL X, double Tol) {
  INTERVAL KX = K(f, X);
  if( Intersection(KX, KX, X) ) {
    if ( Diam(KX) < Tol ) {
      if ( KX < X )
        cout « " Unique root in the interval " « KX « endl;
      else
        cout « "Maybe a root in the interval " « KX « endl;
    }
    else {
      krawczyk(f, INTERVAL(Inf(KX), Mid(KX)), Tol);
      krawczyk(f, INTERVAL(Mid(KX), Sup(KX)), Tol);
    }
  }
}

int main(int argc, char * argv[])
{
  INTERVAL   X(atof(argv[1]), atof(argv[2]));
  double   Tol(atof(argv[3]));

  krawczyk(function, X, Tol);
  return 0;
}
```

Figure 5.6: An recursive Krawczyk/bisection scheme.

```
Domain          : [-10, 10.001]
Tolerance       : 0.0001
Function calls  : 93
 Unique root in the interval [-9.424778381078896,-9.424777714652166]
 Unique root in the interval [-6.283186567789095,-6.283184616617528]
 Unique root in the interval [-3.141592664446003,-3.141592643554570]
 Unique root in the interval [-0.000000642135820,+0.000000633776171]
 Unique root in the interval [+0.739085090377269,+0.739085180166543]
 Unique root in the interval [+3.141585146865033,+3.141614053175048]
 Unique root in the interval [+6.283180355257999,+6.283192959093360]
 Unique root in the interval [+9.424777813489889,+9.424778211462723]
```

5.2 Optimization

In this section, we will focus on the task of optimization. In essence, given a function $f\colon D \to \mathbb{R}$, we wish to find the minimal value y^* of f over the domain D:

$$y^* = y^*(f; D) = \inf\{f(x)\colon x \in D\}, \tag{5.3}$$

as well as the set points in D where this minimum is attained:

$$\mathbb{E}^* = \mathbb{E}^*(f; D) = \{x^* \in D\colon f(x^*) = y^*\}. \tag{5.4}$$

Note that finding the maximal value of a function is achieved by minimizing $-f$. In what follows, we will assume that f is continuous on its domain D, which is assumed to be compact. This implies that $\mathbb{E}^*$ is non-empty, i.e., f attains its minimum on D. Note that the set of extremal points $\mathbb{E}^*$ may very well contain an entire line segment. This happens e.g. when f is constant, and $D = [x]$.

We will employ techniques from interval analysis to enclose both y^* and $\mathbb{E}^*$. Starting with the extremal value y^*, note that (5.3) can be expressed as

$$y^* = \inf\{y\colon y \in R(f; D)\}. \tag{5.5}$$

Therefore, if we can enclose $R(f; D)$ tightly, then we immediately have a tight enclosure of y^*. In fact, optimization is less strenuous than finding a range enclosure, since we need only focus our attention on the lower end-point of the range $R(f; D)$.

The techniques we will employ are very similar to the root-finding techniques presented earlier. By excluding subsets of D, we will attempt to solve the problem

$$f(x) \leq \tilde{y}, \tag{5.6}$$

but with a varying value for $\tilde{y}$, which is the current upper bound for y^*. During the process of solving (5.6) we may come across a smaller value for $\tilde{y}$, in which case its

value is decreased, and the excluding process continues. The goal is to make $\tilde{y}$ as close to y^* as possible, since then solving (5.6) generates the desired set

$$\mathbb{E}^* = \{x^* \in D\colon f(x^*) \leq y^*\} = \{x^* \in D\colon f(x^*) = y^*\}. \tag{5.7}$$

Realistically, we can hope to find an interval $[y^*]$ containing y^*, and an interval enclosure $\cup_i [x_i^*]$ of the set $\mathbb{E}^*$. Following the spirit of [HH95], we will use several criteria for discarding subintervals from our search: the midpoint method, the monotonicity test, and the concavity test. We will from now on assume that the domain D is an interval $[x]$. More complicated domains can be treated by considering unions of intervals.

5.2.1 The midpoint method

To initialize our search, we set $\mathbb{E}^{(0)} = [x^{(0)}] = [x]$, and $\tilde{y}^{(0)} = +\infty$. It is then clear that $y^* \leq \tilde{y}^{(0)}$, and thus that $\mathbb{E}^* \subseteq \{x \in [x]\colon f(x) \leq \tilde{y}^{(0)}\}$.

Now, suppose that we have arrived at stage k of our optimization process. At this point, we have an upper bound for the minimal function function value $y^* \leq \tilde{y}^{(k)}$, and the enclosure

$$\mathbb{E}^* \subseteq \{x \in [x]\colon f(x) \leq \tilde{y}^{(k)}\} \subseteq \mathbb{E}^{(k)} = \bigcup_{i=1}^{N_k} [x_i^{(k)}]$$

of the minimizing set. For each $i = 1, \ldots, N_k$, we do the following: First, we compute the enclosure $[y_i] = F([x_i^{(k)}])$, and check if $\tilde{y}^{(k)} < \underline{y}_i$. If this is the case, then the set $[x_i^{(k)}]$ is eliminated from the search, see Figure 5.7. Otherwise, we continue by computing the midpoint sample $m_i = f(\mathrm{mid}([x_i^{(k)}]))$, which is compared to $\tilde{y}^{(k)}$. If $m_i < \tilde{y}^{(k)}$, we set $\tilde{y} = m_i$. Also, we bisect the subdomain $[x_i^{(k)}]$, and add the two halves to $\mathbb{E}^{(k+1)}$. When we have looped through all i, we have arrived at stage $k+1$, and set $\tilde{y}^{(k+1)} = y^{(k)}$. The entire process is repeated until some stopping criteria has been met. This could be e.g. when the subdomains are sufficiently small, i.e., when

$$\max\left\{\mathrm{rad}\big([\tilde{x}]\big)\colon [\tilde{x}] \in \mathbb{E}^{(k)}\right\} \leq \texttt{TOL}, \tag{5.8}$$

or when the local range enclosures are sufficiently tight, i.e., when

$$\max\left\{\mathrm{rad}\big(F([\tilde{x}])\big)\colon [\tilde{x}] \in \mathbb{E}^{(k)}\right\} \leq \texttt{TOL}. \tag{5.9}$$

Using the stopping criteria (5.8) will produce an upper bound for the global minimum: $y^* \leq \tilde{y} = \tilde{y}^{(k)}$, as well as an enclosure $\mathbb{E}^{(k)}$ of the set $\{x \in [x]\colon f(x) \leq \tilde{y}\}$, which of course encloses $\mathbb{E}^*$. We are, however, not provided with any a priori information regarding the quality of the computed upper bound (i.e., a bound on $y^* - \tilde{y}$). On the other hand, we know that the algorithm will terminate within an explicit number of loops. This choice is suitable when we only need to know that the global

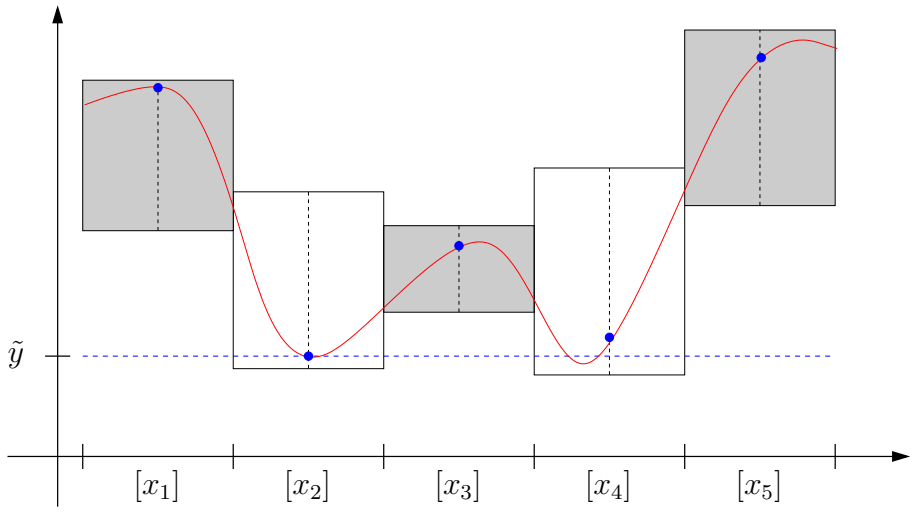


Figure 5.7: Eliminating subintervals $[x_1]$, $[x_3]$, and $[x_5]$ with the midpoint method.

minimum is smaller than some specific value, or that it is attained within some specific set.

Using the stopping criteria (5.9) has the advantage of producing an enclosure of global minimum: $y^* \in [y^*] = [\tilde{y} - \texttt{TOL}, \tilde{y}]$. A disadvantage is that we no longer know that the algorithm will terminate within an explicit number of loops. This choice is suitable when we really need to know the value and location of the global minimum.

We stress that any point $\check{x}_i^{(k)}$ of the subinterval $[x_i^{(k)}]$ could be used for taking the sample $m_i = f(\check{x}_i^{(k)})$. Consequently, we are not worried about the fact that the midpoint of an interval may not be exactly representable: any point within the subinterval will suffice for our needs. In some situations it may be beneficial to sample f at several points of $[x_i^{(k)}]$, and then use their collective minimum as the value of m_i.

The procedure just described can be programmed (in C++[2]) as in Figure 5.8.

Note that the line

```
double midY = Sup(F(Hull(Mid(localX))));
```

computes the largest element of the set $F([\check{x}, \check{x}])$, where $\check{x}$ is the floating point evaluation of the midpoint of `localX`. The reason why we want the *largest* element is that we do not want to underestimate the global minimum.

Identifying $\tilde{y}^{(k)}$ with `globalMin`, and $\mathbb{E}^{(k)}$ with `minimizingList`, the function call

```
INTERVAL Y = boundMinimum(minimizingList, globalMin);
```

[2]Here, and the remaining sections we are using the PROFIL/BIAS package [PrBi].

```
void valGlobMin(INTERVAL domain, double TOL)
{
  double globalMin = Machine::PosInfinity;
  List<INTERVAL> minimizingList, domainList;
  domainList += domain;
  while( !IsEmpty(domainList) ) {
    INTERVAL localX = Pop(domainList);
    INTERVAL localY = F(localX);
    if ( globalMin >= Inf(localY) ) {
      double midY = Sup(F(Hull(Mid(localX))));
      if ( globalMin > midY )
        globalMin = midY;
      if ( Rad(localY) > TOL )              // Or 'Rad(localX) > TOL'.
        splitAndStore(localX, domainList);
      else
        minimizingList += localX;
    }
  }
  INTERVAL Y = boundMinimum(minimizingList, globalMin);
  cout « "Global minimum in: " « Mid(Y) « " +- " « Rad(Y) « endl;
  cout « "Attained within  : " « minimizingList « endl;
}
```

Figure 5.8: A midpoint optimization procedure.

computes the interval $[y] = [z] \cap [-\infty, \tilde{y}^{(k)}]$, where

$$\underline{z} = \min\{\inf\{F([\tilde{x}])\} \colon [\tilde{x}] \in \mathbb{E}^{(k)}\} \quad \text{and} \quad \overline{z} = \min\{\sup\{F([\tilde{x}])\} \colon [\tilde{x}] \in \mathbb{E}^{(k)}\}.$$

Thus the resulting enclosure $[y]$ may be tighter than could be expected by the specified tolerance.

Example 5.2.1 *Running the coded algorithm with* $f(x) = \cos x$ *and* `TOL` $= 2^{-10}$ *over the domain* $[x] = [-15, 15]$ *produces the following output:*

```
Domain           : [-15,15]
Global minimum in: -0.999988347965415 +- 1.16520345848636e-05
Attained within  :
  1: [-9.433593750000000e+00,-9.375000000000000e+00]
  2: [-3.164062500000000e+00,-3.105468750000000e+00]
  3: [+3.105468750000000e+00,+3.164062500000000e+00]
  4: [+9.375000000000000e+00,+9.433593750000000e+00]
Tolerance        : 9.765625000000000e-04  (2^-10)
Function calls   : 122
```

Decreasing the tolerance to 2^{-40} *produces several intervals. Note, however, that the subintervals labeled 2 and 3 are adjacent, as are those labeled 4 and 5. This has to*

do with the symmetry of the function with respect to the bisection points. It is easy to write a small routine that merges adjacent intervals before sending the list to the output.

```
Domain           : [-15,15]
Global minimum in: -0.999999999999998 +- 2.22044604925031e-15
Attained within  :
  1: [-9.424778223037720e+00,-9.424776434898376e+00]
  2: [-3.141594529151917e+00,-3.141592741012573e+00]
  3: [-3.141592741012573e+00,-3.141590952873230e+00]
  4: [+3.141590952873230e+00,+3.141592741012573e+00]
  5: [+3.141592741012573e+00,+3.141594529151917e+00]
  6: [+9.424776434898376e+00,+9.424778223037720e+00]
Tolerance        : 9.094947017729282e-13  (2^-40)
Function calls   : 343
```

It should be pointed out that decreasing the tolerance is meaningless once the maximum accuracy of the floating point computation of f is reached. Repeating Example 5.2.1 with tolerance 2^{-50} produces 16 boxes, and with tolerance 2^{-51}, the program never terminates!

Exercise 5.2.2 *Write a program implementing the midpoint method, but use the stopping criteria (5.9). How does decreasing the tolerance now affect the output?*

Exercise 5.2.3 *Write and add the merging routine described in Example 5.2.1 to your program.*

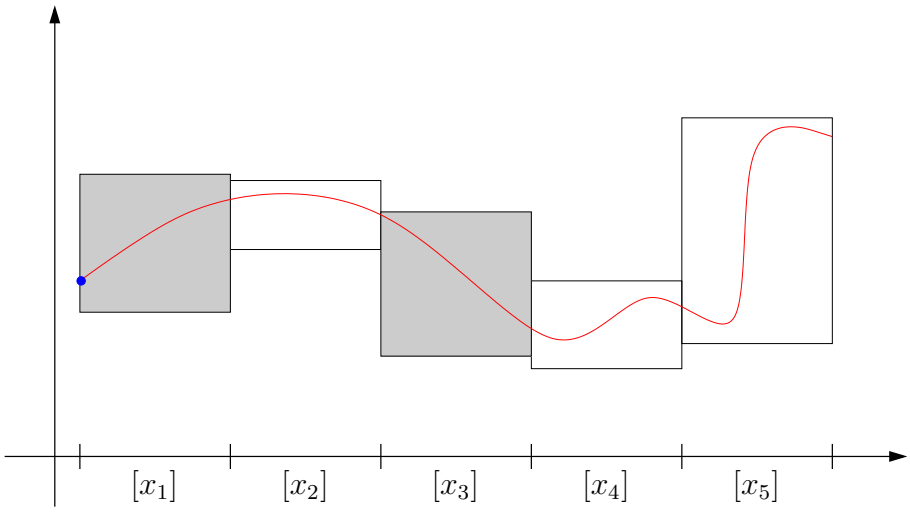


Figure 5.9: Eliminating subinterval $[x_3]$, and shrinking $[x_1]$ to its left endpoint with the monotonicity test.

5.2.2 The monotonicity test

Checking if f is monotone on subintervals is also an effective way of detecting areas where the global minimum cannot be obtained. This requires that we have an interval extension of f', which we will call F'. If we, at some stage of our search, have a subinterval $[x_i^{(k)}] \in \mathbb{E}^{(k)}$ such that $0 \in F'([x_i^{(k)}])$, then we bisect $[x_i^{(k)}]$, and add the two halves to $\mathbb{E}^{(k+1)}$. If, on the other hand, $0 \notin F'([x_i^{(k)}])$, there are two situations that need separate treatment. In the first case, $[x_i^{(k)}]$ belongs to the interior of $[x^{(0)}]$, in which case we can delete the entire set $[x_i^{(k)}]$ from our search. In the second case, $[x_i^{(k)}]$ has at least one endpoint in common with $[x^{(0)}]$. If f is increasing on $[x_i^{(k)}]$, and if $\underline{x}_i^{(k)} = \underline{x}^{(0)}$, then we shrink $[x_i^{(k)}]$ to the point $\underline{x}_i^{(k)}$, which is added to $\mathbb{E}^{(k+1)}$. If f is decreasing on $[x_i^{(k)}]$, and if $\overline{x}_i^{(k)} = \overline{x}^{(0)}$, then we shrink $[x_i^{(k)}]$ to the point $\overline{x}_i^{(k)}$, which is added to $\mathbb{E}^{(k+1)}$. A typical application of the monotonicity test is illustrated in Figure 5.9.

Implementing this test is straight-forward. Below a simple `C++` routine is presented that preserves, shrinks or deletes a subinterval (denoted `oldX` in the code), depending on the monotonicity properties of f.

```
action monotoneTest(INTERVAL &newX, INTERVAL oldX, INTERVAL domain)
{
  INTERVAL df = DF(X);
  if ( Mig(df) == 0 ) {
    newX = oldX;
    return PRESERVE;
  }
  else if ( (Inf(oldX) == Inf(domain)) && (0.0 < Inf(df)) )
    newX = Hull(Inf(oldX));
  else if ( (Sup(oldX) == Sup(domain)) && (Sup(df) < 0.0) )
    newX = Hull(Sup(oldX));
  else
    return DELETE;
  return SHRINK;
}
```

This small routine is merged with `valGlobMin` as follows:

```
...
  while( !IsEmpty(domainList) ) {
    INTERVAL oldX = First(domainList);
    RemoveCurrent(domainList);
    INTERVAL localX;
    if ( DELETE == monotoneTest(localX, oldX, domain) )
      continue;
    INTERVAL localY = F(localX);
    if ( globalMin >= Inf(localY) ) {
...
```

When the test deletes a subinterval, the instruction `continue` returns the program flow to the top of the `while`-loop, i.e., a new element of `domainList` is examined.

Example 5.2.4 *Continuing Example 5.2.1, we repeat the runs, but this time using the additional monotonicity test.*

Running the updated algorithm with $f(x) = \cos x$ *and* `TOL` $= 2^{-10}$ *over the domain* $[x] = [-15, 15]$ *produces the following output:*

```
Domain           : [-15,15]
Global minimum in: -0.999988347965415 +- 1.16520345848636e-05
Attained within  :
  1: [-9.433593750000000e+00,-9.375000000000000e+00]
  2: [-3.164062500000000e+00,-3.105468750000000e+00]
  3: [+3.105468750000000e+00,+3.164062500000000e+00]
  4: [+9.375000000000000e+00,+9.433593750000000e+00]
Tolerance        : 9.765625000000000e-04  (2^-10)
Function calls   : 76
Derivative calls : 63
```

Without the monotonicity test, 122 *interval function calls were required. This should*

be compared to the $76 + 63 = 139$ *calls now required. With the monotonicity test, however, decreasing the tolerance to* 2^{-40} *does not produce additional intervals.*

```
Domain           : [-15,15]
Global minimum in: -0.999999999999998 +- 2.22044604925031e-15
Attained within  :
  1: [-9.424778223037720e+00,-9.424776434898376e+00]
  2: [-3.141592741012573e+00,-3.141590952873230e+00]
  3: [+3.141590952873230e+00,+3.141592741012573e+00]
  4: [+9.424776434898376e+00,+9.424778223037720e+00]
Tolerance        : 9.094947017729282e-13  (2^-40)
Function calls   : 196
Derivative calls : 183
```

Again, the number of calls $196+183 = 379$ *is slightly larger than the number required* (343) *without the monotonicity test. An advantage is that we now obtain a tighter enclosure of the minimizing set* $\mathbb{E}^*$.

5.2.3 The convexity test

Digging even deeper for helpful information, the second derivative f'' also provides a powerful means of detecting areas where the global minimum cannot be obtained. This, of course, requires that we have an interval extension of f'', which we will call F''. If the global minimum is attained within the interior of $[x^{(0)}]$, i.e., if $\mathbb{E}^* \overset{\circ}{\subset} [x^{(0)}]$, then for all $x^* \in \mathbb{E}^*$, we have $f''(x^*) \geq 0$. Therefore, if we, at some stage of our search, have a subinterval $[x_i^{(k)}] \in \mathbb{E}^{(k)}$ such that $\sup\{F''([x_i^{(k)}])\} < 0$, then $[x_i^{(k)}]$ can be deleted from the search, unless $[x_i^{(k)}]$ has a boundary point in common with $[x^{(0)}]$. If the latter happens, we shrink $[x_i^{(k)}]$ to the set $[x_i^{(k)}] \cap \{\underline{x}^{(0)}, \overline{x}^{(0)}\}$, which is added to $\mathbb{E}^{(k+1)}$. A typical application of the convexity test is illustrated in Figure 5.10.

```
action convexTest(INTERVAL &newX, const INTERVAL &oldX, const INTERVAL &domain)
{
  INTERVAL ddf = DDF(oldX);
  if ( Sup(ddf) < 0.0 ) {
    if ( Inf(oldX) == Inf(domain) )
      newX = Hull(Inf(oldX));
    else if ( Sup(oldX) == Sup(domain) )
      newX = Hull(Sup(oldX));
    else
      return DELETE;
    return SHRINK;
  }
  return PRESERVE;
}
```

Much like the monotonicity test, it is very easy to implement the necessary checks

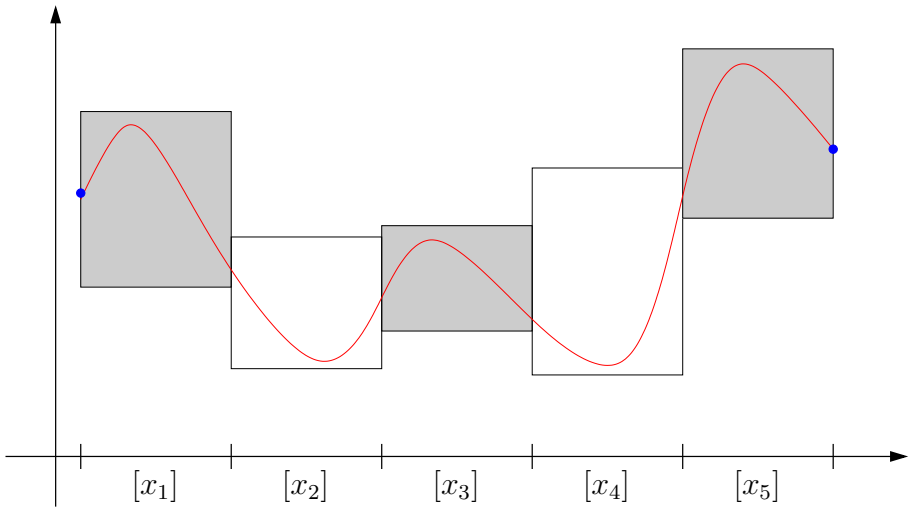


Figure 5.10: Shrinking the subintervals $[x_1]$ and $[x_5]$, and eliminating $[x_3]$ with the convexity test.

for convexity. The core `C++` routine eiter preserves, shrinks or deletes a subinterval, depending on the convexity properties of f.

Exercise 5.2.5 *Modify your code so that it also implements the convexity test. On termination, the program should print the percentages of the initial interval deleted by the various tests. Note that this will depend on which order they are performed.*

Exercise 5.2.6 *Assuming that f is twice continuously differentiable, it is possible to employ the interval Newton methods for f' on subintervals where f'' does not vanish. This will determine if f has a stationary point or not in the subinterval. Modify your code so that it implements this additional test.*

5.3 Quadrature

We now turn our attention to the problem of computing, or estimating, definite integrals, i.e., integrals of the type

$$I = \int_a^b f(x)dx \tag{5.10}$$

Today there exists a myriad of numerical methods designed to produce efficient and accurate estimates of integrals of type (5.10). We will briefly mention a few of the most elementary such methods: the *midpoint* method, the *trapezoid* method, and *Simpson's* method.

All integration methods rely on the basic fact that the integral operator is additive with respect to the domain, i.e., for any partition $a = x_0 \leq x_1 \leq \cdots \leq x_N = b$, we have

$$\int_a^b f(x)dx = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)dx.$$

By taking a sufficiently fine partition, it is likely that the integrand f can be well-approximated by low order polynomials p_i over each subinterval $[x_{i-1}, x_i]$, which results in the approximation

$$\begin{aligned} I = \int_a^b f(x)dx &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)dx \\ &\approx \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} p_i(x)dx = \sum_{i=1}^{N} \big(P_i(x_i) - P_i(x_{i-1})\big). \end{aligned} \tag{5.11}$$

Here, the polynomials P_i satisfy the relation $P_i'(x) = p_i(x)$.

For the midpoint method, we take $p_i(x) = f(\frac{x_{i-1}+x_i}{2})$, i.e., on each subinterval the corresponding p_i is a constant equal to the function value at the midpoint of $[x_{i-1}, x_i]$, the latter which we will denote by $\tilde{x}_i$. The trapezoid method is obtained by taking $p_i(x) = f(x_{i-1})(1 - s_i(x)) + f(x_i)s_i(x)$ with $s_i(x) = (x - x_{i-1})/(x_i - x_{i-1})$, i.e., on each subinterval the corresponding p_i is the linear function passing from $f(x_{i-1})$ to $f(x_i)$. Both methods are illustrated in Figure 5.11.

Simpson's method approximates f, restricted to $[x_{i-1}, x_i]$, with a second degree polynomial that coincides with f at the points[3] x_{i-1}, $\tilde{x}_i$, and x_i.

Assuming that the partition is uniform, i.e., that $x_i = a + ih$, where $h = (b - a)/N$

[3] Our definition is not entirely standard. It is more common to sample at three consecutive grid points x_{i-1}, x_i, and x_{i+1}. This, however, requires that N is even.

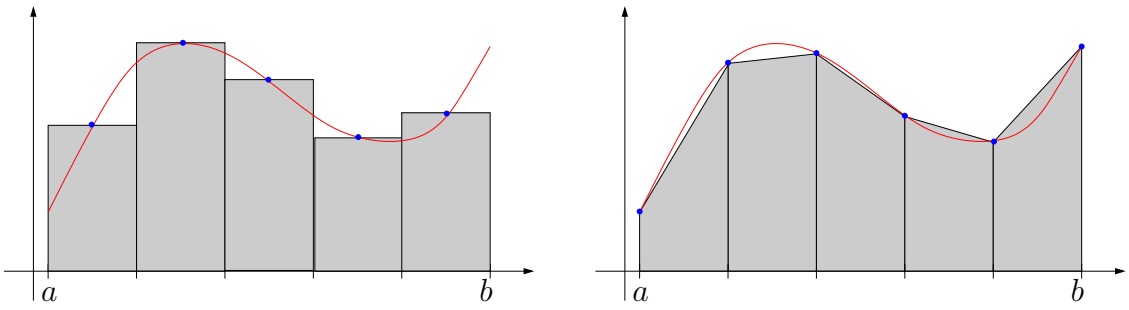


Figure 5.11: (a) The midpoint method. (b) The trapezoidal method.

is the step-size, there are elegant formulas for the methods just described.

$$
\begin{aligned}
I &\approx M_a^b(f,N) \stackrel{\text{def}}{=} h\sum_{i=1}^{N} f(\tilde{x}_i) \\
I &\approx T_a^b(f,N) \stackrel{\text{def}}{=} h\left(\frac{f(x_0)}{2} + \sum_{i=1}^{N-1} f(x_i) + \frac{f(x_N)}{2}\right) \\
I &\approx S_a^b(f,N) \stackrel{\text{def}}{=} \frac{1}{3}T_a^b(f,N) + \frac{2}{3}M_a^b(f,N).
\end{aligned}
\tag{5.12}
$$

The code in Figure 5.12 illustrates how easily one can implement the three methods.

It is not hard to obtain explicit formulas for the integral errors, assuming that the integrand is sufficiently smooth ($f \in C^2([a,b])$ for the midpoint and trapezoidal methods, and $f \in C^4([a,b])$ for Simpson's method). Rather surprisingly, it turns out that the midpoint method is twice as accurate as the trapezoidal rule (for sufficiently small h). More precisely, for some $\zeta_M, \zeta_T, \zeta_S \in [a,b]$, we have

$$
\begin{aligned}
\int_a^b f(x)dx - M_a^b(f,N) &= \frac{b-a}{24}h^2 f''(\zeta_M) \stackrel{\text{def}}{=} \Delta M_a^b(f,N), \\
\int_a^b f(x)dx - T_a^b(f,N) &= -\frac{b-a}{12}h^2 f''(\zeta_T) \stackrel{\text{def}}{=} \Delta T_a^b(f,N), \\
\int_a^b f(x)dx - S_a^b(f,N) &= -\frac{b-a}{2880}h^4 f^{(4)}(\zeta_S) \stackrel{\text{def}}{=} \Delta S_a^b(f,N).
\end{aligned}
\tag{5.13}
$$

Of course, the points $\zeta_M, \zeta_T, \zeta_S$ are unknown. Nevertheless, if we can somehow bound the derivatives appearing in (5.13) over the domain $[a,b]$, we obtain upper bounds on the errors. In the next section, this type of approach will be studied closer.

Example 5.3.1 *Using the methods described in this section, let us approximate the integral* $I = \int_{-2}^{2} f(x)dx$, *where* $f(x) = \sin(\cos(e^x))$. *For convenience, we will take* $N = 2^i$ *for* $i = 0, \ldots, 10$. *The results are presented in Table 5.3.1. From an*

```
#include <iostream>
using namespace std;
typedef double (*pfcn)(double);

double midpoint(pfcn f, double a, double b, int N) {
  double h   = (b - a)/N;
  double sum = f(a + 0.5*h);
  for (int i = 1; i < N; i++)
    sum += f(a + (i + 0.5) * h);
  return h*sum;
}

double trapezoid(pfcn f, double a, double b, int N) {
  double h   = (b - a)/N;
  double sum = (f(a) + f(b))*0.5;
  for (int i = 1; i < N; i++)
    sum += f(a + i*h);
  return h*sum;
}

double simpsons(pfcn f, double a, double b, int N) {
  return (2*midpoint(f, a, b, N) + trapezoid(f, a, b, N))/3;
}

double integrand(double x) { return sin(cos(exp(x))); }

int main(int argc, char * argv[])
{
  double a(atof(argv[1])), b(atof(argv[2]));
  int    N(atoi(argv[3]));

  cout «  midpoint(integrand, a, b, N) « endl;
  cout « trapezoid(integrand, a, b, N) « endl;
  cout «  simpsons(integrand, a, b, N) « endl;

  return 0;
}
```

Figure 5.12: A straight-forward C++ implementation of the integration schemes.

Table 5.1: Integral estimates of $\int_{-2}^{2} \sin(\cos(e^x))dx$.

N	$M^2_{-2}(f,N)$	$T^2_{-2}(f,N)$	$S^2_{-2}(f,N)$	$\Delta M^b_a(f,N)$	$\Delta T^b_a(f,N)$	$\Delta S^b_a(f,N)$
1	2.0575810	2.5399605	2.2183742	$-7.189 \cdot 10^{-1}$	$-1.201 \cdot 10^{0}$	$-8.797 \cdot 10^{-1}$
2	0.0257958	2.2987708	0.7834541	$+1.313 \cdot 10^{0}$	$-9.601 \cdot 10^{-1}$	$+5.552 \cdot 10^{-1}$
4	1.2556376	1.1622833	1.2245195	$+8.303 \cdot 10^{-2}$	$+1.764 \cdot 10^{-1}$	$+1.141 \cdot 10^{-1}$
8	1.4089676	1.2089605	1.3422986	$-7.030 \cdot 10^{-2}$	$+1.297 \cdot 10^{-1}$	$-3.630 \cdot 10^{-3}$
16	1.3527757	1.3089641	1.3381718	$-1.411 \cdot 10^{-2}$	$+2.970 \cdot 10^{-2}$	$+4.969 \cdot 10^{-4}$
32	1.3425891	1.3308699	1.3386827	$-3.920 \cdot 10^{-3}$	$+7.799 \cdot 10^{-3}$	$-1.398 \cdot 10^{-5}$
64	1.3396402	1.3367295	1.3386699	$-9.715 \cdot 10^{-4}$	$+1.939 \cdot 10^{-3}$	$-1.235 \cdot 10^{-6}$
128	1.3389108	1.3381848	1.3386688	$-2.421 \cdot 10^{-4}$	$+4.839 \cdot 10^{-4}$	$-8.039 \cdot 10^{-8}$
256	1.3387292	1.3385478	1.3386687	$-6.046 \cdot 10^{-5}$	$+1.209 \cdot 10^{-4}$	$-5.068 \cdot 10^{-9}$
512	1.3386838	1.3386385	1.3386687	$-1.511 \cdot 10^{-5}$	$+3.022 \cdot 10^{-5}$	$-3.174 \cdot 10^{-10}$
1024	1.3386725	1.3386612	1.3386687	$-3.778 \cdot 10^{-6}$	$+7.556 \cdot 10^{-6}$	$-1.988 \cdot 10^{-11}$

auxillary computation, we happen to know that $I \in 1.33866870740^{20}_{18}$. *This allows us to compute the integration errors, which are also displayed in the table.*

Our theory is clearly confirmed by the numerics: Asymptotically, each time we double N, *we see both the midpoint and trapezoidal errors decrease by a factor four. Also, the simpson error decreases by a factor 16, as predicted by (5.13). We also note that the trapeziodal error is roughly twice as large as the midpoint error, in good agreement with (5.13).*

5.3.1 Enclosure methods

Interval analysis provides an elegant means for approximating definite integrals. If f admits a well-defined interval extension F over the integration domain $[a,b]$, then, by Theorem 3.1.11, we have the naive enclosure

$$I \in w([a,b])F([a,b]),$$

where we let $w([a,b]) = b - a$ denote the *width* of an interval. Of course, in most cases this will produce a terrible estimate of I: the resulting interval will most likely be very wide. As mentioned earlier, the integral operator is additive with respect to the domain. Thus, given any partition $a = x_0 \leq x_1 \leq \cdots \leq x_N = b$, we can decompose the integration domain into the N non-overlapping subintervals

$$[a,b] = [x_0, x_1] \cup [x_1, x_2] \cup \cdots \cup [x_{N-1}, x_N]$$

and bound the range of f over each subinterval separately. This produces the enclosure

$$I = \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)dx \in \sum_{i=1}^{N} w([x_{i-1}, x_i])F([x_{i-1}, x_i]). \tag{5.14}$$

```
#include <iostream>
#include "Interval.h"
#include "Functions.h"
using namespace std;
typedef INTERVAL (*pfcn)(INTERVAL);

INTERVAL integrate(pfcn f, INTERVAL x, int N) {
  INTERVAL dX     = INTERVAL(0, Diam(x))/N; // Get (over)estimated grid size.
  INTERVAL localX = Inf(x);                 // Start at the left endpoint.
  INTERVAL quad   = 0;                      // Initialize the accumulated quad.
  for (int i = 0; i < N; i++) {             // Work through all grids.
    localX = Sup(localX) + dX;              // Guard against domain overlap.
    if ( Intersection(localX, localX, x) )  // Guard against domain overflow.
      quad += f(localX)*diam(localX);       // Use an interval-valued diameter.
  }
  return quad;
}

INTERVAL integrand(INTERVAL X) { return sin(cos(exp(x))); }

int main()
{
  INTERVAL X(atof(argv[1]), atof(argv[2]));
  int      N(atoi(argv[3]));

  cout « integrate(integrand, X, N) « endl;
  return 0;
}
```

Figure 5.13: A simple-minded enclosure method.

If the integrand f is Lipschitz on $[a, b]$, then we can use Theorem 3.1.15, and deduce that the width of the enclosure (5.14) is at most proportional to $\max_i w([x_{i-1}, x_i])$.

The most straight-forward approach is to split the domain of integration into N equally wide subintervals: we set $h = (b-a)/N$ and $x_i = a + ih$, $i = 0, \ldots, N$. This produces the enclosure

$$I_a^b(f, N) \stackrel{\text{def}}{=} h \sum_{i=1}^{N} F([x_{i-1}, x_i]), \tag{5.15}$$

which satisfies $w(I_a^b(f, N)) = \mathcal{O}(1/N)$.

Example 5.3.2 *Approximate the integral* $I = \int_{-2}^{2} f(x)dx$, *where* $f(x) = \sin(\cos(e^x))$ *by computing the enclosures* $I_{-2}^{2}(f, N)$ *for* $N = 1, 100, 10000,$ *and* 1000000.

Note that, starting from $N = 100$, *we clearly see the reciprocal correspondance between the number of subintervals and the enclosure widths.*

Table 5.2: Integral enclosures of $\int_{-2}^{2} \sin(\cos(e^x))dx$ using (5.15).

N	$I^2_{-2}(f,N)$	$w(I^2_{-2}(f,N))$
10^0	$[-3.36588, 3.36588]$	$6.73177 \cdot 10^0$
10^2	$[1.26250, 1.41323]$	$1.50729 \cdot 10^{-1}$
10^4	$[1.33791, 1.33942]$	$1.50756 \cdot 10^{-3}$
10^6	$[1.33866, 1.33868]$	$1.50758 \cdot 10^{-5}$

From this example, it should be clear that method (5.15) is *absolutely not* the right way to go about computing integral enclosures. The enclosure method (5.15) requires *one million* function evaluations to match the accuracy obtained with a mere 1024 evaluations using the trapezoid method (5.12). This is to be expected, as the trapezoid method has a quadratic error term, whereas the error of our proposed enclosure method (5.15) is linear.

In order to improve the accuracy of our integral enclosures, we must generate tighter bounds on the integrand. This can be achieved by employing the techniques of automatic differentiation, described in Chapter 4. From now on, we will assume that the integrand f is n times continuously differentiable over the domain of integration: $f \in C^n([x])$. Then, using the notation $f_k(\tilde{x}) = f^k(\tilde{x})/k!$, we can Taylor expand f around the midpoint $\tilde{x} = \text{mid}([x])$:

$$f(x) = \sum_{k=0}^{n-1} f_k(\tilde{x})(x-\tilde{x})^k + f_n(\zeta_x)(x-\tilde{x})^n. \tag{5.16}$$

Here, the point $\zeta_x \in [x]$ is usually unknown. We can enclose the remainder term appearing in (5.16) by first computing $F_n([x])$, and then forming

$$\varepsilon_n = \text{mag}\big(F_n([x]) - f_n(\tilde{x})\big),$$

where we recall that $\text{mag}([a]) = \max\{|\underline{a}|, |\overline{a}|\}$. This produces the enclosure of the integrand

$$f(x) \in \sum_{k=0}^{n} f_k(\tilde{x})(x-\tilde{x})^k + [-\varepsilon_n, \varepsilon_n]|x-\tilde{x}|^n, \tag{5.17}$$

valid for all $x \in [x]$. We are now prepared to compute the integral itself:

$$\begin{aligned}\int_{\tilde{x}-r}^{\tilde{x}+r} f(x)dx &\in \int_{\tilde{x}-r}^{\tilde{x}+r} \left(\sum_{k=0}^{n} f_k(\tilde{x})(x-\tilde{x})^k + [-\varepsilon_n, \varepsilon_n]|x-\tilde{x}|^n \right) dx \\ &= \sum_{k=0}^{n} f_k(\tilde{x}) \int_{-r}^{r} x^k dx + [-\varepsilon_n, \varepsilon_n] \int_{-r}^{r} |x|^n dx.\end{aligned}$$

Table 5.3: Integral enclosures of $\int_{-2}^{2} \sin(\cos(e^x))dx$.

N	$E^2_{-2}(f,6,N)$	$w(E^2_{-2}(f,6,N))$
9	$[0.86325178469, 1.81128961988]$	$9.4804 \cdot 10^{-1}$
12	$[1.28416304745, 1.39316025451]$	$1.0900 \cdot 10^{-1}$
21	$[1.33783795371, 1.33950680633]$	$1.6689 \cdot 10^{-3}$
75	$[1.33866863493, 1.33866878008]$	$1.4514 \cdot 10^{-7}$

Now something interesting happens. Since we are integrating monomials over a domain centered at the origin, there is a lot of cancellation: all odd-numbered terms will evaluate to zero. Continuing our calculations, we have

$$\begin{aligned}\int_{\tilde{x}-r}^{\tilde{x}+r} f(x)dx &\in \sum_{k=0}^{n} f_k(\tilde{x}) \int_{-r}^{r} x^k dx + [-\varepsilon_n, \varepsilon_n] \int_{-r}^{r} |x|^n dx \\ &= \sum_{k=0}^{\lfloor n/2 \rfloor} f_{2k}(\tilde{x}) \int_{-r}^{r} x^{2k} dx + [-\varepsilon_n, \varepsilon_n] \int_{-r}^{r} |x|^n dx \\ &= 2 \left(\sum_{k=0}^{\lfloor n/2 \rfloor} f_{2k}(\tilde{x}) \frac{r^{2k+1}}{2k+1} + [-\varepsilon_n, \varepsilon_n] \frac{r^{n+1}}{n+1} \right).\end{aligned}$$

For large domains of integration, we can form a suitably fine partition $a = x_0 \leq x_1 \leq \cdots \leq x_N = b$, and form the enclosure

$$\begin{aligned}\int_a^b f(x)dx &= \sum_{i=1}^{N} \int_{x_{i-1}}^{x_i} f(x)dx = \sum_{i=1}^{N} \int_{\tilde{x}_i - r_i}^{\tilde{x}_i + r_i} f(x)dx \\ &\in 2 \sum_{i=1}^{N} \left(\sum_{k=0}^{\lfloor n/2 \rfloor} f_{2k}(\tilde{x}_i) \frac{r_i^{2k+1}}{2k+1} + [-\varepsilon_{n,i}, \varepsilon_{n,i}] \frac{r_i^{n+1}}{n+1} \right). \qquad (5.18)\end{aligned}$$

In Figure 5.14, we present a simple `C++` implementation of algorithm (5.18) that employes a uniform partition, i.e., $x_i = a + ih$, where $h = (b-a)/N$ is the step-size. In Table 5.3, we present the outcome of some computations using the same piece of code, with Taylor expansions of degree six.

5.3.2 Adaptive integration

Of course, it may be wasteful to split the domain of integration into a uniform grid. A subinterval $[x_i]$ that produces a very narrow Riemann term

$$T_n(f, [x_i]) = 2 \left(\sum_{k=0}^{\lfloor n/2 \rfloor} f_{2k}(\tilde{x}_i) \frac{r_i^{2k+1}}{2k+1} + [-\varepsilon_{n,i}, \varepsilon_{n,i}] \frac{r_i^{n+1}}{n+1} \right)$$

```
#include <iostream>
#include "taylor.h"
using namespace std;
typedef taylor (*pfcn)(const taylor &);

INTERVAL riemannTerm(pfcn f, INTERVAL x, int Deg) {
  INTERVAL mid = INTERVAL(Mid(x));
  INTERVAL rad = INTERVAL(diam(x))/2;       // Use an interval-valued diameter.
  taylor   fx  = f(variable(mid, Deg));     // The 'thin' Taylor series.
  INTERVAL sum = fx[0]*rad;
  for (int k = 2; k <= Deg; k += 2)
    sum += fx[k]*Power(rad, k + 1)/(k + 1);
  taylor Fx  = f(variable(x, Deg));         // The 'fat' remainder term.
  double eps = Abs(Fx[Deg] - fx[Deg]);
  sum += INTERVAL(-eps, +eps)*Power(rad, Deg + 1)/(Deg + 1);
  return 2*sum;
}

INTERVAL integrate(pfcn f, INTERVAL x, int Deg, int N) {
  INTERVAL dX     = INTERVAL(0, Diam(x))/N;
  INTERVAL localX = Inf(x);
  INTERVAL quad   = 0;
  for (int i = 0; i < N; i++) {
    localX = Sup(localX) + dX;              // Guard against domain overlap.
   if ( Intersection(localX, localX, x) )   // Guard against domain overflow.
     quad += riemannTerm(f, localX, Deg);
  }
  return quad;
}

taylor integrand(const taylor &x) { return sin(cos(exp(x))); }

int main(int argc, char * argv[])
{
  INTERVAL   X(atof(argv[1]), atof(argv[2]));
  int      Deg(atoi(argv[3]));              // Degree of Taylor series.
  int        N(atoi(argv[4]));              // Number of cells.

  cout « integrate(integrand, X, Deg, N);
  return 0;
}
```

Figure 5.14: An implementation of the Taylor-based enclosure method (5.18).

Table 5.4: Adaptive integral enclosures of $\int_{-2}^{2} \sin(\cos(e^x))dx$.

TOL	$A^2_{-2}(f, 6, \texttt{TOL})$	$w(A^2_{-2}(f, 6, \texttt{TOL}))$	$N_{\texttt{TOL}}$
10^{-1}	$[1.33229594606, 1.34500942603]$	$1.2713 \cdot 10^{-2}$	9
10^{-2}	$[1.33822575109, 1.33911045235]$	$8.8470 \cdot 10^{-4}$	12
10^{-4}	$[1.33866170207, 1.33867571626]$	$1.4014 \cdot 10^{-5}$	21
10^{-8}	$[1.33866870618, 1.33866870862]$	$2.4304 \cdot 10^{-9}$	75

need not be further decomposed. Instead, all efforts should be concentrated where the terms are relatively wide.

An adaptive integration scheme can easily be attained by recusively bisecting the original domain into a collection of subintervals $\{[x_i]\}_{i=1}^{N_{\texttt{TOL}}}$ whose corresponding Riemann terms meet the tolerance requirement:

$$w\big(T_n(f, [x_i])\big) \leq \texttt{TOL}([x_i]) \stackrel{\text{def}}{=} \texttt{TOL}([a, b])\frac{w([x_i])}{b-a}.$$

It is clear that the recursive subdivision must terminate after a finite number of steps. By (5.18) it follows that $w\big(T_n(f, [x_i])\big) = \mathcal{O}(w([x_i])^{n+1})$. If the integrand f is of class C^{n+1}, then $f^{(n)}$ is of class C^1, and by Theorem 3.1.15, there exist positive real numbers K_i (that are uniformly bounded) such that $\varepsilon_{n,i} \leq K_i w([x_i])$. Therefore, we actually have

$$w\big(T_n(f, [x_i])\big) = \mathcal{O}(w([x_i])^{n+2}),$$

which means that the tolerance requirement will be satisfied after a finite number of subdivisions, even for the case $n = 0$, corresponding to the naive enclosure method (5.14).

In Figure 5.15, we present a straight-forward `C++` implementation of the adaptive scheme just outlined.

Example 5.3.3 *Continuing from Example 5.3.2, let us now use the adaptive integration scheme described above. We will compute the interval enclosures using the tolerances* $\texttt{Tol} = 10^{-1}, 10^{-2}, 10^{-4}$, *and* 10^{-8}. *The order of the Taylor expansions is set to six.*

Here we have also listed the number of subintervals $N_{\texttt{TOL}}$ *produced by the adaptive scheme. Comparing with Table 5.3, we see the advantage of adaptively decomposing the integration domain. Also see Figure 5.16, illustrating the successive enclosures together with the corresponding partitions of the domain of integration.*

Exercise 5.3.4 *Note that the final enlosure widths in Table 5.4 are quite a bit smaller than requested. Can you explain why? How could one fix this?*

```
#include <iostream>
#include "taylor.h"
using namespace std;
typedef taylor (*pfcn)(const taylor &);

INTERVAL riemannTerm(pfcn f, INTERVAL X, int Deg) {
  INTERVAL mid = INTERVAL(Mid(X));
  INTERVAL rad = INTERVAL(diam(X))/2;       // Use an interval-valued diameter.
  taylor   fx  = f(variable(mid, Deg));     // The 'thin' Taylor series.
  INTERVAL sum = fx[0]*rad;
  for (int k = 2; k <= Deg; k += 2)
    sum += fx[k]*Power(rad, k + 1)/(k + 1);
  taylor Fx  = f(variable(X, Deg));         // The 'fat' remainder term.
  double eps = Abs(Fx[Deg] - fx[Deg]);
  sum += INTERVAL(-eps,+eps)*Power(rad, Deg + 1)/(Deg + 1);
  return 2*sum;
}

INTERVAL integrate(pfcn f, INTERVAL X, int Deg, double Tol) {
  INTERVAL sum = riemannTerm(f, X, Deg);
  if ( Diam(sum) <= Tol )
    return sum;
  else
    return integrate(f, INTERVAL(Inf(X), Mid(X)), Deg, Tol/2) + \
           integrate(f, INTERVAL(Mid(X), Sup(X)), Deg, Tol/2);
}

taylor integrand(const taylor &x) { return sin(cos(exp(x))); }

int main(int argc, char * argv[])
{
  INTERVAL   X(atof(argv[1]), atof(argv[2]));
  int      Deg(atoi(argv[3]));              // Degree of Taylor series.
  double   Tol(atof(argv[4]));              // Error tolerance.

  cout « integrate(integrand, X, Deg, Tol);
  return 0;
}
```

Figure 5.15: An adaptive version of algorithm (5.18).

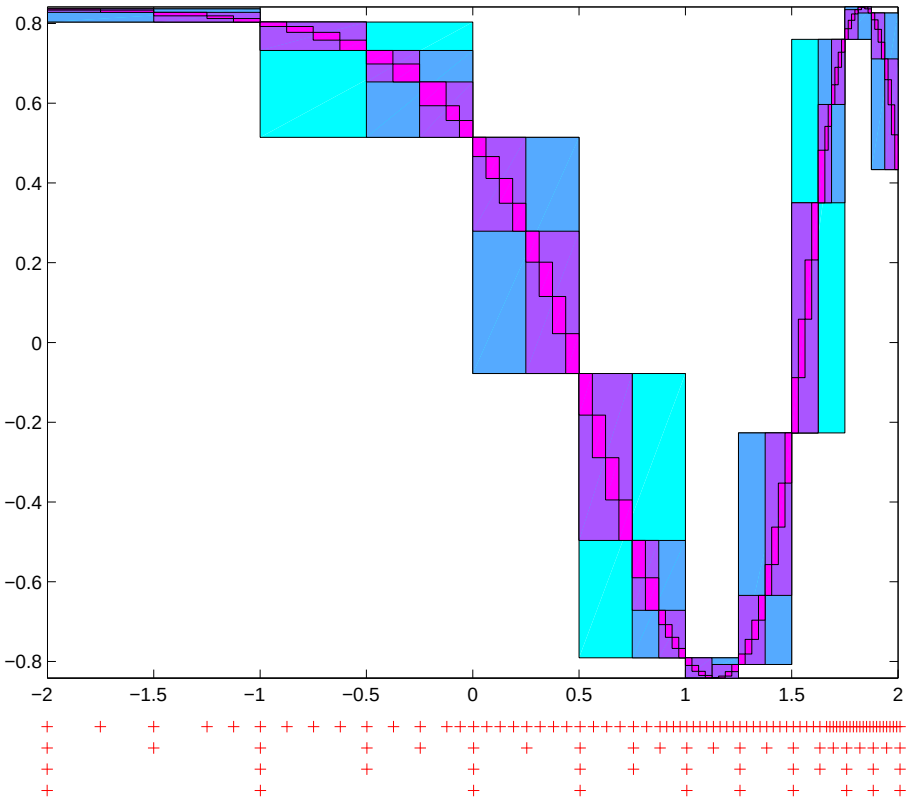


Figure 5.16: Refined enclosures of the integrand $f(x) = \sin(\cos(e^x))$.

5.4 Computer Lab IV

Problem 1. Write an interval bisection routine such as described in Section 5.1.1. Your program should prompt the user for the search domain $[x]$, and a tolerance `TOL`. What are the results you get for the function $f(x) = \sin x - \cos x$ with $[x] = [-10, +10]$ and $\texttt{TOL} = 0.0001$?

Problem 2. Write an interval Newton routine. Your program should prompt the user for the search domain $[x]$. What are the results you get for the function $f(x) = \sin(\cos(x-3))$ with the search domains $[x_1] = [1, 2]$, $[x_2] = [1.5, 2.5]$, and $[x_3] = [2, 3]$?

Problem 3. Combine the ideas from the two previous problems and write a hybrid bisection/interval Newton routine. It should bisect the search domain into subintervals $[x_k]$ until either $\text{diam}([x_k]) \leq \texttt{TOL}$, or $0 \notin F'([x_k])$. In the latter case the subinterval should undergo an interval Newton search. Your program should prompt the user for the search domain $[x]$, and a tolerance `TOL`. What are the results you get for the function $f(x) = \sin(\cos(x-3))$ with the search domain $[x] = [-10, 10]$, and $\texttt{TOL} = 0.001$?

Problem 4. Write an interval optimizer, as described in Section 5.2. You may choose to use any or all of the midpoint, monotonicity, or convexity checks. Your program should prompt the user for the search domain $[x]$, and a tolerance `TOL`. What are the results you get for the function

$$f(x) = x^2 - \frac{1}{2}e^{-(a(x-\frac{1}{2}))^2}$$

with $a = 10000$, $[x] = [-10, +10]$ and $\texttt{TOL} = 10^{-10}$?

Problem 5. Write a program that computes rigorous enclosures of the standard quadrature methods by computing the approximations (5.12) in interval arithmetic, and by enclosing the remainder terms (5.13) using taylor series arithmetic (from Problem 4 of Computer Lab III).

Problem 6. Write a simple-minded interval integrator (as described in Section 5.3.1), and redo Example 5.3.1 from the notes.

Problem 7. Write an adaptive, taylor-based, interval integrator (as described in Section 5.3.2), and redo Example 5.3.3.

Chapter 6

Ordinary differential equations

In this chapter, we will describe various procedures for enclosing the solution to an ordinary differential equation (ODE).

6.1 A gentle mathematical introduction

In the real-valued setting, a general initial-value problem can be formulated as follows: find a differentiable function $x\colon [0,T] \to \mathbb{R}$ that satisfies the following ODE:

$$\begin{cases} \dot{x}(t) = f(x(t), t) \\ x(0) = x_0. \end{cases} \tag{6.1}$$

Here $f\colon D \times [0,T] \to \mathbb{R}$ is called a *vector field*, and is assumed to satisfy a Lipschitz condition in its first variable, i.e., there is a positive constant K such that, for all $t \in [0,T]$ and for all $x_1, x_2 \in D$, we have

$$|f(x_1,t) - f(x_2,t)| \leq K|x_1 - x_2|.$$

With this restriction on f, it is possible to prove that there exists a unique solution to (6.1) for every initial value $x_0 \in D$. The path defined by such a solution curve $x(t)$ is sometimes called an *orbit* or a *trajectory*. Given an initial condition $x_0 \in D$, it is not clear how long time its solution exists, i.e., we do not know the smallest value of t for which $x(t)$ is not well-defined. If we take the initial value x_0 very close to the boundary of the domain D, then its trajectory may leave the D in a very short time, after which we really have no control of the solution. On the other hand, if we restrict the initial values to a proper sub-domain $D' \subset D$, it is possible to obtain a lower bound T' on how long time the solution exists (and is unique).

Exercise 6.1.1 *Given the ODE (6.1), where f has the Lipschitz constant K, find a minimal time of existence T' in terms of D and D'. You may assume that both domains are compact intervals.*

It is common to describe the solution of (6.1) as a *flow*. A flow is simply a solution that depends explicitly on both the x and the t variables, and is denoted $\varphi(x,t)$. It is defined by

$$\begin{cases} \frac{d}{dt}\varphi(x,t) = f(\varphi(x,t),t) \\ \varphi(x,0) = x. \end{cases} \tag{6.2}$$

Thus the trajectory passing through the point x at time $t = 0$ will be at position $\varphi(x,t)$ at time t.

Example 6.1.2 *Consider the ODE $\dot{x} = -tx$. The solution with initial condition $x(0) = x_0$ is given by $x(t) = x_0 e^{-t^2/2}$ (verify this!). The corresponding flow is $\varphi(x,t) = xe^{-t^2/2}$. Note that the solutions are well-defined for all $t \in \mathbb{R}$.*

Now consider the ODE $\dot{x} = x^2$. The solution with initial condition $x(0) = x_0 \neq 0$ is given by $x(t) = (1/x_0 - t)^{-1}$ (verify this!). The corresponding flow is $\varphi(x,t) = (1/x - t)^{-1}$. This time, the solutions are well-defined only for $t \in (-\infty, 1/x_0)$. If $x_0 = 0$, then $x(t) = \varphi(0,t) = 0$ for all $t \in \mathbb{R}$.

Our aim is to find an interval extension Ψ of the real-valued flow φ such that, for all $[t] \subseteq [0,T']$, and $[x] \subseteq D'$, we have the enclosure

$$R(\varphi; [x] \times [t]) \subseteq \Psi([x],[t]). \tag{6.3}$$

In particular, we should have $x_0 \in [x_0]$, $t \in [0,T'] \Rightarrow \varphi(x_0,t) \in \Psi([x_0],t)$.

6.2 Simple enclosure methods

To begin with, we will restrict our attention to the *autonomous* case, in which the vector field f does not explicitly depend on the variable t. The general ODE can then be formulated as

$$\begin{cases} \dot{x}(t) = f(x(t)) \\ x(0) = x_0. \end{cases} \tag{6.4}$$

We will use the fact that the (6.4) combined with (6.2) can be expressed as the integral equation

$$\varphi(x_0,t) = x_0 + \int_0^t f(\varphi(x,s))ds. \tag{6.5}$$

Now, fixing the set of possible initial values $x_0 \in [x_0]$ for the moment, suppose that we have a rough enclosure $\varphi(x_0,t) \in \Psi(t)$ for all $(x_0,t) \in [x_0] \times [0,T']$. Then we certainly have the inclusion

$$\varphi(x_0,t) \in \Psi(0) + \int_0^t F(\Psi(s))ds, \tag{6.6}$$

where F is an interval extension of f. The interval-valued integral is evaluated by forming increasingly finer interval Riemann sums, and taking the limit[1]. We can now form the sequence of interval enclosures

$$\begin{aligned} \Psi^{(1)}(t) &= \Psi(t), \\ \Psi^{(k+1)}(t) &= \Psi(0) + \int_0^t F(\Psi^{(k)}(s))ds. \end{aligned} \tag{6.7}$$

Using the Lipschitz property of f, we can prove that $\mathrm{rad}\big(R(f;[x])\big) \leq K\mathrm{rad}([x])$ for all $[x] \in D$. Thus, assuming that F also is Lipschitz[2] on D, we have $\mathrm{rad}(F([x])) \leq K'\mathrm{rad}([x])$ for some positive K'. Therefore, we also have

$$\begin{aligned} \mathrm{rad}\big(\Psi^{(k+1)}(t)\big) &\leq \mathrm{rad}\big(\Psi(0)\big) + \int_0^t K'\mathrm{rad}\big(\Psi^{(k)}(s)\big)ds \\ &\leq \mathrm{rad}\big(\Psi(0)\big) + K't \max\left\{\mathrm{rad}\big(\Psi^{(k)}(s)\big) : s \in [0,t]\right\}. \end{aligned}$$

Thus, if t is chosen small enough to guarantee that $K't < 1$, the sequence of enclosures $\{\Psi^{(k)}\}_{k=1}^{\infty}$ will be nested and converge to a tube containing the true solution. Due to the inherent properties of interval arithmetic, however, the radius of the tube is bounded from below by the quantity

$$r_{\min}(t) = \frac{\mathrm{rad}\big(\Psi(0)\big)}{1 - K't}. \tag{6.8}$$

The tube, of course, is nothing else than the graph of the interval function $\Psi^*(t)$ defined by

$$\Psi^*(t) = \Psi(0) + \int_0^t F(\Psi^*(s))ds.$$

In fact, $\Psi^*(t)$ is the unique fixed point of the contraction mapping P defined by

$$P(\Xi(t)) = \Psi(0) + \int_0^t F(\Xi(s))ds,$$

and acting on the space of continuously differentiable functions.

In the absolutely simplest setting, we take all enclosures to be independent of the time variable t, i.e., we want to construct $\Psi^{(k)}(t) = [z^{(k)}]$, $k \in \mathbb{Z}^+$, where each $[z^{(k)}]$ is an interval. We initialize the process by setting $\Psi^{(1)}(t) = [z^{(1)}]$, where $[z^{(1)}]$ contains the initial value x_0. Having done so, it is possible to compute the largest time T' such that, for all $t \in [0, T']$, the new variant of the integral equation (6.7),

$$[z^{(k+1)}] = [x_0] + \int_0^t F([z^{(k)}])ds = x_0 + [0,t] \times F([z^{(k)}]), \tag{6.9}$$

[1] Taking the limit is not strictly necessary, since every finite interval Riemann sum contains the true integral.

[2] An interval-valued function F is *Lipschitz* if there is a positive K such that for all $[x]$ we have $\mathrm{rad}\,(F([x])) \leq K\mathrm{rad}\,([x])$.

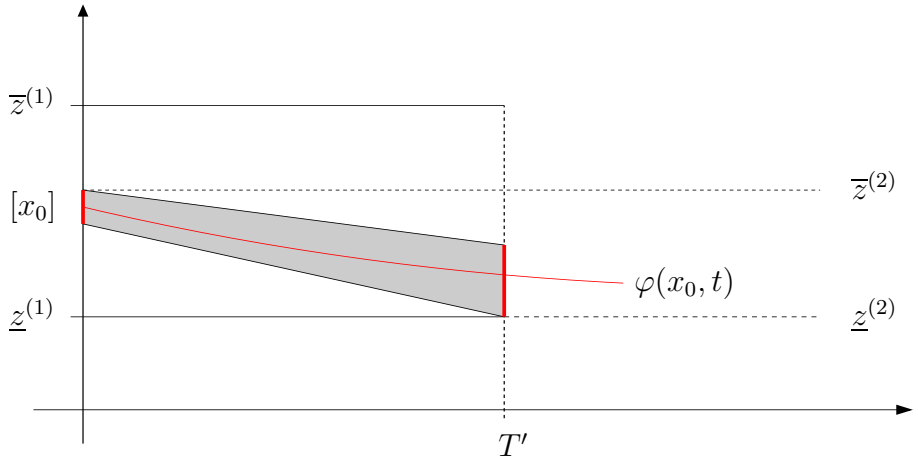


Figure 6.1: Computing successively tighter enclosures of the flow.

produces a nested sequence of intervals. It follows that any real solution starting from $[x_0]$ does not leave any of the sets $[z^{(k)}]$ before time T', i.e., we can enclose the trajectory as follows:

$$R\big(\varphi;[x_0]\times[0,T']\big)\in[z^{(k)}]\qquad k=1,2,\ldots. \tag{6.10}$$

Just looking at a snapshot of the enclosures of the flows starting from $[x_0]$, i.e., fixing a time $t\in[0,T']$, we have the possibly much tighter enclosures

$$\varphi(x_0,t)\in R(\varphi(\cdot,t);[x_0])\subseteq[x_0]+tF([z^{(k)}]),\qquad k=1,2,\ldots. \tag{6.11}$$

When we vary the parameter t, each of these enclosures form a truncated cone $C^{(k)}(t)$ extending from $[x_0]$. The rectangular hull of each cone $C^{(k)}(T')$ is contained in $[z^{(k)}]$, see Figure 6.1.

With the widths of the solution enclosures $C^{(k)}(t)$ strictly increasing with time, it is fair to wonder whether we can ever obtain realistic results in a situation where e.g. all solutions are contracted toward a single trajectory. This is exactly what happens for solutions of $\dot{x}=-tx$, which we solved in Example 6.1.2. Independently of the initial value x_0, all trajectories approach the constant solution $x(t)=0$ at an exponential rate. In particular, if we follow an entire interval of initial conditions $[x_0]$ along the flow, its image is contracted as t increases. In this specific case we have an explicit formula for the image: $R\big(\varphi(\cdot,t);[x_0]\big)=[x_0]e^{-t^2/2}$. Recall that, in view of (6.8) an enclosure $\Psi^{(k)}(t)$ of the flow of $[x_0]$ cannot have a radius smaller than

$$\frac{\operatorname{rad}\big(\Psi(0)\big)}{1-K't}\geq\frac{\operatorname{rad}\big([x_0]\big)}{1-K't}.$$

A successful way around this discouraging fact is to consider the endpoints of $[x_0]$ separately. By the uniqueness property of the solutions, we know that for all $t\in[0,T']$ and $x\in[x_0]$, we have the enclosure

$$\varphi(\underline{x}_0,t)\leq\varphi(x,t)\leq\varphi(\overline{x}_0,t).$$

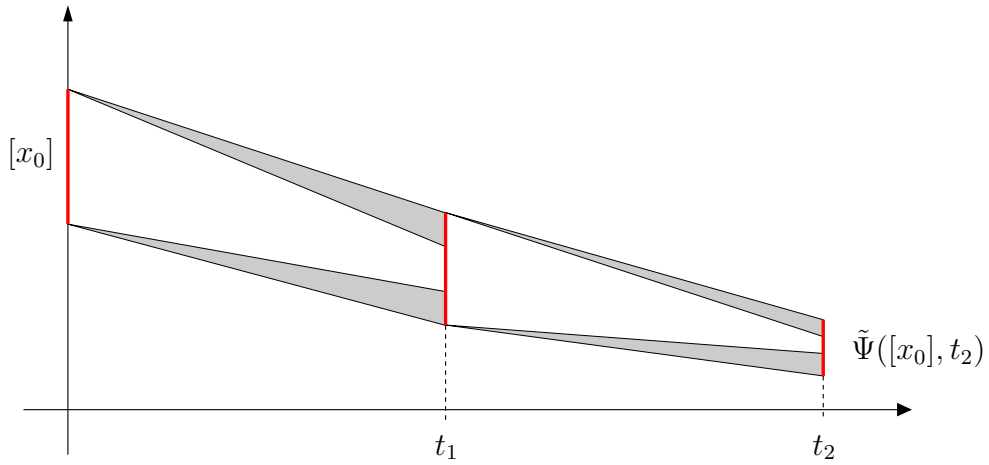


Figure 6.2: Obtaining global contraction with expanding local enclosures.

Therefore, it suffices to compute enclosures of the two trajectories starting from the thin sets $\underline{x}_0$ and $\overline{x}_0$, respectively. A typical situation is illustrated in Figure 6.2. Here we see that, although each enclosure is strictly increasing, their combined bounds form a contracting set.

Also note that, if at some point in time t_1, the enclosures have become too wide for our likings, we simply merge the two enclosures to form a new interval

$$[x_1] = \Psi(\underline{x}_0, t_1) \sqcup \Psi(\overline{x}_0, t_1) \stackrel{\text{def}}{=} \tilde{\Psi}([x_0], t_1)$$

The entire procedure is then repeated starting from with $[x_1]$ instead of $[x_0]$, producing

$$[x_2] = \Psi(\underline{x}_1, t_2 - t_1) \sqcup \Psi(\overline{x}_1, t_2 - t_1) \stackrel{\text{def}}{=} \tilde{\Psi}([x_0], t_2),$$

and so on.

It should be pointed out that this approach only works in the one-dimensional setting. In higher dimensions, we cannot enclose the flow of an initial box by a finite number of trajectories, unless the solutions satisfy some monotonicity properties, see [Tu02].

Given an initial set $[x_0]$ and a positive time $T < T'$, we can obtain arbitrarily tight enclosures of the exact range $R(\varphi(\cdot, T); [x_0])$. This is achieved by splitting the interval $[0, T]$ into several sub-intervals $0 = t_0 < t_1 < t_2 < \cdots < t_n = T$, and performing the entire procedure described above on each sub-interval $[t_i, t_{i+1}]$. The number of nodes t_i is determined by the desired accuracy, see Figure 6.3

Exercise 6.2.1 *Write a program implementing the ideas from this section. Once a crude enclosure has been established, there are many different ways one can improve the enclosures (e.g. using centered forms, checking for monotonicity etc). Add some of your own improvements to the code.*

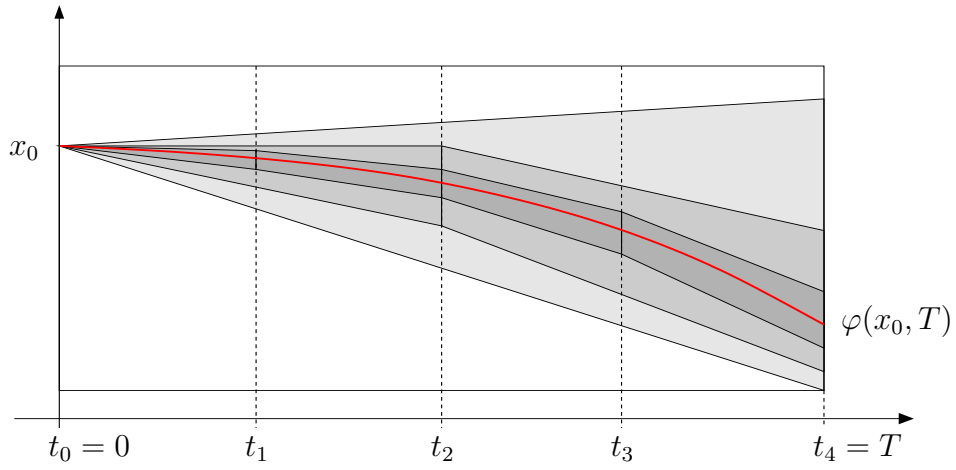


Figure 6.3: Increasingly tight enclosures of the trajectory $\varphi(x_0, t)$.

Exercise 6.2.2 *What changes must be made to allow for non-autonomous differential equations $\dot{x} = f(x, t)$?*

The methods presented in this section are *first-order* methods. As a consequence, the condition $K't < 1$ heavily restricts the maximal time steps allowed. Higher order methods (e.g. Taylor series methods of a higher degree) partially overcome this problem, but the area is still an active field of research, see e.g., [Mo66], [BM98] or [NJ01].

6.3 High-order methods

One of the most powerful features of automatic differentiation is that it provides us with an elegant and effective means of computing numerical solutions to initial-value problems of the type $\phi'(t) = f(\phi(t))$, with initial condition $\phi(t_0) = \phi_0$.

Indeed, expressing ϕ as its Taylor series centered at t_0, we have

$$\phi(t) = \phi_0 + \phi_1(t - t_0) + \cdots + \phi_n(t - t_0)^n + \ldots,$$

and by taking the formal derivative of ϕ, we find the relation

$$(\phi')_k = (k + 1)\phi_{k+1} \qquad k \geq 0.$$

Combining this with the fact that $\phi' = f(\phi)$, we get the elegant recursive formula for the Taylor coefficients of the solution ϕ:

$$\phi_{k+1} = \frac{f(\phi)_k}{k + 1} \quad (k > 0). \tag{6.12}$$

Here, the value of the nominator can be obtained by the techniques presented in Chapter 4. As a consequence, by using automatic differentiation, we can numerically

solve the differential equation $\phi'(t) = f(\phi(t))$; $\phi(t_0) = \phi_0$ by a taylor method of any desired order. This should be viewed in stark contrast to e.g. the Runge-Kutta methods, each of which has a fixed order of convergence. No longer are we restricted to adaptively changing the step-size of the numerical method, we can now also vary the *order* of the method. Naturally, this makes the solvers more robust and accurate than their fixed-order variants.

Exercise 6.3.1 *Write a simple implementation of an solver for initial value problems, using the relation (6.12). Test your module on the problem* $\phi'(t) = \sin(1 + \cos t)$; $\phi(0) = 1$ *by computing approximations of* $\phi(t)$, $t \in [0, 1]$ *using orders* $1, 2, \ldots, 10$, *and some suitable fixed step-size. It may be informative to plot the approximating polynomials.*

6.4 Rigorous high-order examples

In what follows, we will incorporate remainder bounds into the Taylor series method method, thus making it rigorous. By the mean value theorem, we have

$$\phi(x, t + h) = \phi_0(x, t) + \phi_1(x, t)h + \cdots + \phi_n(x, t)h^n + \phi_{n+1}(x, t)s^{n+1},$$

for some $s \in [t, t + h]$. Note that the Taylor coefficients $\phi_k = \phi_k(x, t)$ of the flow ϕ can be computed according to (6.12). Combining the ideas presented in Section 6.2 with the use of Taylor series enables us to enclose a trajectory of a scalar ODE with a high-degree method.

Assume that we are at stage k, i.e., we have rigorously computed $[x_k]$ which is a rigorous enclosure of $\phi([x_0], t_k)$. Now we want to proceed to step $k + 1$. We will, once again, use the fact that we are working in one phase variable, and treat the endpoints of $[x_k]$ separately, similarly to Figure 6.3.

Using e.g. the first-order method (6.9), we start by computing a feasible flow time $h_k = t_{k+1} - t_k$, and the associated crude enclosures $[\hat{z}_k]$ and $[\check{z}_k]$ of the two endpoints' flows during this time. The flow time is obtained by taking the minimum of the the computed flow times for both endpoints of $[x_k]$. Next, we compute

$$[\hat{w}_{k+1}] = \phi_0(\overline{x}_k, t_k) + \phi_1(\overline{x}_k, t_k)h_k + \cdots + \phi_n(\overline{x}_k, t_k)h_k^n + \phi_{n+1}([\hat{z}_k], t_k)[0, h_k]^{n+1},$$
$$[\check{w}_{k+1}] = \phi_0(\underline{x}_k, t_k) + \phi_1(\underline{x}_k, t_k)h_k + \cdots + \phi_n(\underline{x}_k, t_k)h_k^n + \phi_{n+1}([\check{z}_k], t_k)[0, h_k]^{n+1}.$$

Note that all "thin" Taylor coefficients must be computed separately from the two "wide" coefficients $\phi_{n+1}([\hat{z}_k], t_k)$ and $\phi_{n+1}([\check{z}_k], t_k)$. The latter are computed over the first-order enclosures $[\hat{z}_k]$ and $[\check{z}_k]$, which make them of inferior quality compared to the coefficients computed over point domains. Nevertheless, these "wide" coefficients are scaled by the (very small) interval factor $[0, h_k]^{n+1}$. This allows for a tight enclosure of the image of each endpoint of $[x_k]$ under the flow.

As a final step, we take the hull of both enclosures:

$$[x_{k+1}] = [\check{w}_{k+1}] \sqcup [\hat{w}_{k+1}].$$

This completes one full integration step.

Exercise 6.4.1 *The method just described can easily be modified to allow for non-autonomous ODEs, i.e., problems where the vector field may depend explicitly on the time variable: $\phi'(x,t) = f(\phi(x,t),t)$. Can you see how?*

Let us consider some simple examples of initial value problems to see how the interval solver works. All computations were done on a 2.6 GHz processor.

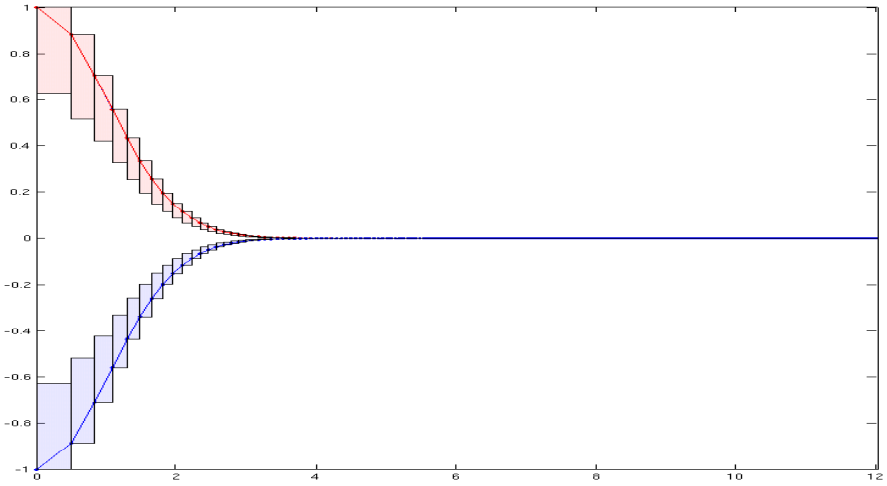


Figure 6.4: Third order interval solution to (6.13) with $[x_0] = [-1,1]$, up to $t = 12$. The *red* boxes are the verification boxes for the upper endpoints and the *blue* boxes are the verification boxes for the lower endpoints. The red and blue lines are linear interpolations of the upper, and lower endpoints, respectively. The interval solution contracts rapidly around $x = 0$. The diameter of the interval solution at time $t = 12$ is less than 10^{-17}. The computational time was 50 seconds.

Example 6.4.2 *As a first example, let us consider the non-autonomous system*

$$\begin{cases} x'(t) = -tx(t), & t \geq t_0 \\ x(t_0) = x_0 \in [-1,1] \end{cases} \tag{6.13}$$

which has the exact solution,

$$x(t) = x_0 e^{-(t^2 - t_0^2)/2}.$$

From this it is clear that $x \equiv 0$ is asymptotically stable. We present the enclosures to the problem (6.13) in Figure 6.4.

Example 6.4.3 *The autonomous system,*

$$\begin{cases} x'(t) = x(t)^2, \quad t \geq 0 \\ x(0) = x_0 \in [1, 1.25] \end{cases} \tag{6.14}$$

has the solution

$$x(t) = \frac{1}{1/x_0 - t}, \quad t \in [0, 1/x_0)$$

which blows up at time $t_c = 1/x_0$. *We present the enclosures to the problem (6.14) in Figure 6.5.*

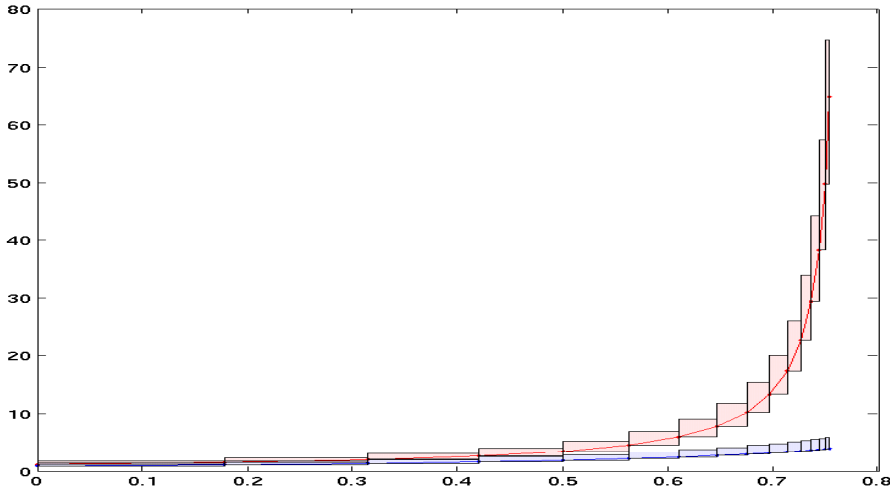


Figure 6.5: Third order interval solution to (6.14) with $[x_0] = [1, 1.25]$, up to $t = 0.75$. The diameter of the upper verification boxes grow rapidly as t approaches $t_c = 0.8$. The computational time was 4 seconds.

Example 6.4.4 *Consider the autonomous ODE,*

$$\begin{cases} x'(t) = -x(t)^3, \quad t \geq 0 \\ x(0) = 1 \end{cases} \tag{6.15}$$

We present the enclosures to the problem (6.15) in Figure 6.6.

Example 6.4.5 *Consider the following ODE,*

$$\begin{cases} x'(t) = x(t)(x(t) - 1), \quad t \geq 0 \\ x(0) = 1 \end{cases} \tag{6.16}$$

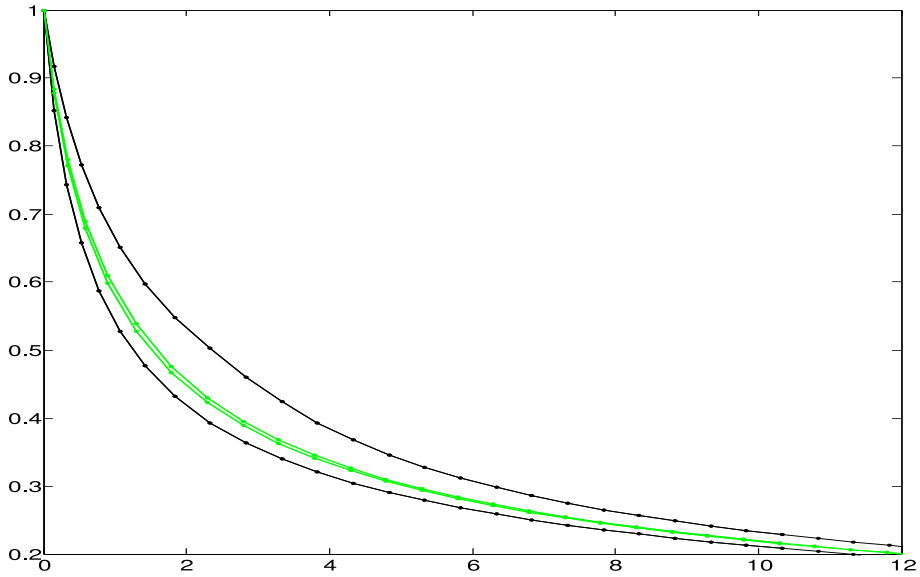


Figure 6.6: A 1st order interval enclosure (black) and a thinner 5th order interval enclosure (green) to (6.15). The verification boxes are not shown. The maximal diameter is 0.1 for the the 1st order enclosure, and 0.01 for the 5th order enclosure. The computational times were 2 and 14 seconds, respectively.

which has the solution $x(t) = 1$, $t \geq 0$. *We compute interval enclosures of (6.16) using 3,5 and 7 degree polynomial approximations. We stop the computation when the diameter of the enclosure goes above* $\texttt{diam}_{\texttt{max}} \doteq 12$. *We present the enclosures to the problem (6.16) in Figure 6.7. Note that the 3rd order solution was stopped at* $t = 6$, *the 5th order at* $t = 9$, *and the 7th order solution was stopped at* $t = 12$.

Example 6.4.6 *Let us consider the following, non-autonomous ODE:*

$$\begin{cases} x'(t) = 5 + \sin t - x(t) & t \geq 1 \\ x(1) = [4, 6]. \end{cases} \tag{6.17}$$

For a 3rd order Taylor method, the diameter of the enclosure at the time point $t = 10$ *is 0.0004, and the computational time is 8 seconds. For a 6th order Taylor method, the diameter decreased to 0.0002, and the computational time increases to 25 seconds. We present the enclosures to the problem (6.17) in Figure 6.8.*

Example 6.4.7 *Consider the, somewhat complicated, ODE*

$$\begin{cases} x'(t) = f(t, x) & t \in [0, 10] \\ x(0) = [3, 3 + 2\varepsilon_{\mathrm{M}}] \end{cases} \tag{6.18}$$

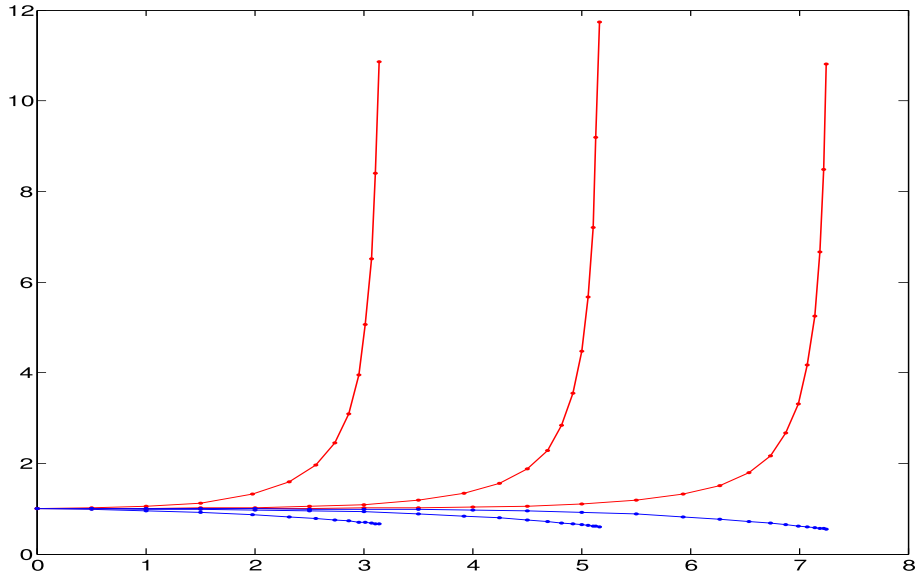


Figure 6.7: Enclosures to (6.16) using orders 3, 5 and 7.

where

$$f(t,x) = (\exp(\exp(-tx)) + 0.01x^3 + 0.1x + 2 + 10\cos(x) + 4\sin(t) - \log(x))/\ldots \\ (0.02x^3 + 4x^2 + 3x + 4 + (x+1)^{0.75}0.001\sin(1.5tx) + 0.001\cos(3.14t))$$

We present the enclosures to the problem (6.18) in Figure 6.9. Table 6.4 shows the maximal diameter of the enclosures and the computational times for different polynomial degree solutions.

degree	**maximal diameter**	**computational time**
1	0.2	10 sec
2	0.07	30 sec
3	0.03	50 sec
10	0.015	10 min
15	0.009	30 min

Table 6.1: Maximal diameter and computational time for different polynomial degree solutions to (6.18).

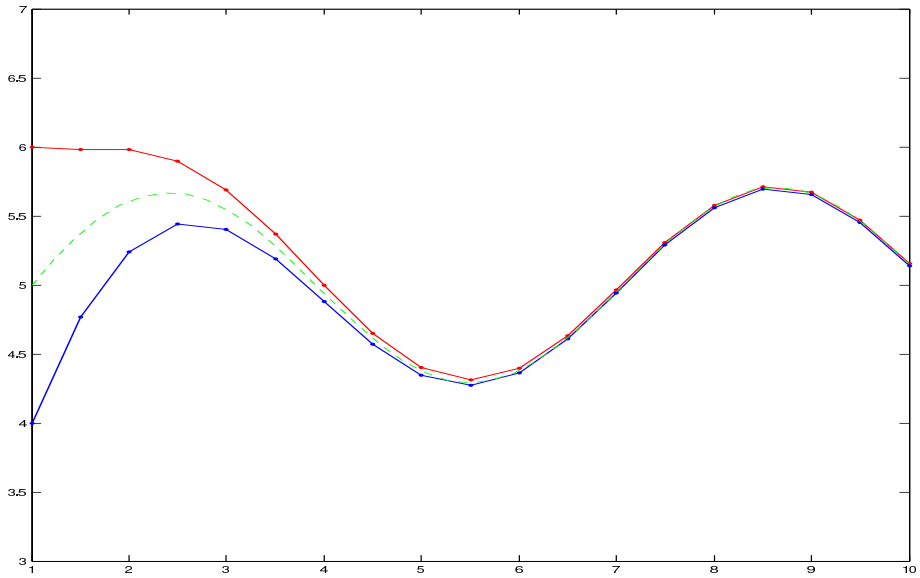


Figure 6.8: Computed enclosures of (6.17) with $t_0 = 1$, $[x_0] = [4, 6]$. The dotted green line is the `MATLAB ode45` solution with initial condition $x_0 = 5$

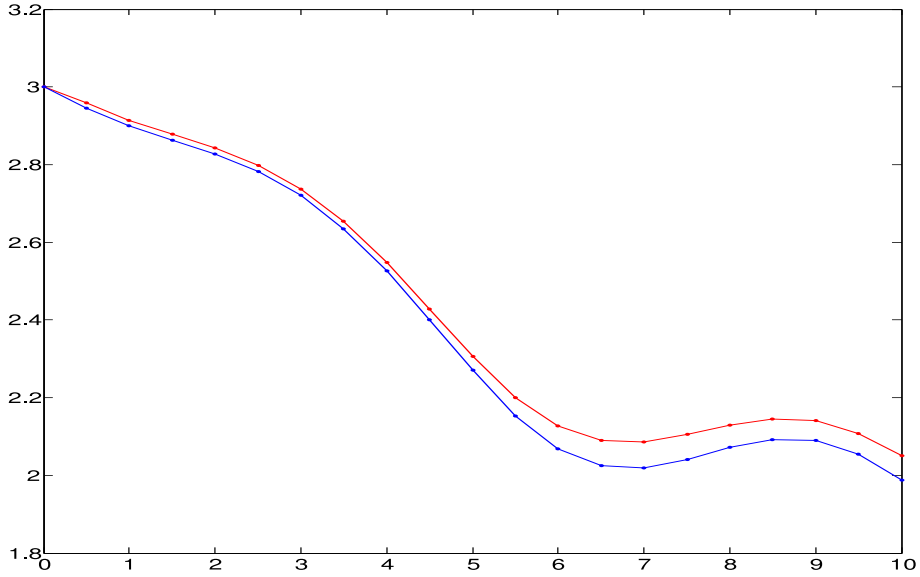


Figure 6.9: The 2nd order enclosure of (6.18)

Bibliography

[Ab88] Aberth, O., Precise Numerical Analysis. Wm. C. Brown Publishers, Dubuque, 1988.

[Ab98] Aberth, O., Precise Numerical Methods Using `C++`. Academic Press, New York, 1998.

[AH83] Alefeld, G., Herzberger, J., Introduction to Interval Computations. Academic Press, New York, 1983.

[Ba22] Banach, S., *Sur les Opérations dans les Ensembles Abstraits et leur Application aux Équations Intégrales*, Fundamenta Mathematicæ **3** (1922), 133-181.

[Be97] Berz, M., *From Taylor Series to Taylor Models*, in "Beam Stability and Nonlinear Dynamics", AIP Conference Proceedings **405** (1997).

[BM98] Berz, M., Makino, K., *Verified Integration of ODEs and Flows using Differential Algebraic Methods on High-Order Taylor Models*, Reliable Computing **4** (1998), 361-369.

[BS97] Bendtsen, C., Stauning, O., *TADIFF, A Flexible C++ Package for Automatic Differentiation*, Technical Report, IMM-REP-1997-07, Lyngby, 1997.

[Br10] Brouwer, L. E. J., *Über Abbildung von Mannigfaltigkeiten*, Mathematische Annalen **71** (1910), 97-115.

[Co77] Corliss, G., *Which Root Does the Bisection Algorithm Find?*, SIAM Review **19** (1977), 325-327.

[CV01] Cuyt, A., Verdonk, B., Kuterna, P., *A Remarkable Example of Catastrophic Cancellation Unraveled*, Computing **66** (2001), 309-320.

[CXSC] CXSC – C++ eXtension for Scientific Computation, version 2.0. Available from `http://www.math.uni-wuppertal.de/org/WRST/xsc/cxsc.html`

[Dw51] Dwyer, P. S. Linear Computations, J. Wiley, New York, 1951.

[EW02] Eugene, L., Wallster, W. G., *Rump's Example Revisited*, Reliable Computing **8** (2002), 245-248.

[FS97] de Figueiredo, L. H., Stolfi, J., Métodos numéricos auto-validados e aplicações. Braz. Math. Colloq. **21**, IMPA, Rio de Janeiro, 1997.

[Ga96] Garnatz, P. G., *CRAY T90 Series IEEE Floating Point Migration Issues and Solutions*, CUG 1996 Spring Proceedings (1996), 345-349.

[GM03] Gabai, D., Meyerhoff, G.R., Thurston, N., *Homotopy hyperbolic 3-manifolds are hyperbolic*, Annals of Mathematics, **157:2** (2003), 335-431.

[Go91] Goldberg, D., *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, Computing Surveys **23:1** (1991), 5-48.

[Gr00] Griewank, A., Evaluating Derivatives. SIAM, Philadelphia, 2000.

[Ha65] Hansen, E., *Interval Arithmetic in Matrix Computations*, J. SIAM Numer. Anal., Ser. B **2:2** (1965), 308-320.

[HW04] Hansen, E., Wallster, G., W., Global Optimization using Interval Analysis (second edition). Marcel Dekker, Inc., New York, 2004.

[Ha95] Hass, J., Hutchings, M., Schlafly, R. *The double bubble conjecture*, Electronic Research Announcements of the AMS **1** (1995), 98-102.

[HH95] Hammer, R. et. al., C++ toolbox for Verified Computing. Springer-Verlag, Berlin, 1995.

[Hi96] Higham, N. J., Accuracy and Stability of Numerical Algorithm. SIAM, Philadelphia, 1996.

[Hi76] Hille, E., Ordinary Differential Equations in the Complex Domain. John Wiley & Sons, Inc., New York, 1976.

[IE85] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std 754-1985, 1985.

[IE87] IEEE Standard for Radix-Independent Floating-Point Arithmetic. ANSI/IEEE Std 854-1987, 1987.

[INv4] INTLAB – INTerval LABoratory, version 4.1.2. Available from `http://www.ti3.tu-harburg.de/~rump/intlab/`

[Kn98] Knuth, D. E., The Art of Computer Programming. Addison-Wesley, 1998.

[Ko02] Koren, I., Computer Arithmetic Algorithms. A K Peters Ltd, 2002.

[KM81] Kulisch, U. W., Miranker, W. L., Computer Arithmetic in Theory and Practice. Academic Press, 1981.

[LT06] Lerch, M., Tischler, G., Wolff von Gudenberg, J. *FILIB++, a fast interval library supporting containment computations*, ACM Trans. Math. Software **32:2** (2006) 299–324.

[Mi65] Milnor, J.W., Topology from the Differentiable Viewpoint. Princeton University Press, Princeton, New Jersey, 1965.

[Mo59] Moore, R. E., Automatic error analysis in digital computation. Technical Report Space Div. Report LMSD84821, Lockheed Missiles and Space Co., 1959.

[Mo65] Moore, R. E., *The Automatic Analysis and Control of Error in Digital Computations Based on the Use of Interval Numbers*, in Error in Digital Computing, Vol 1, John Wiley & Sons Inc, New York, 1965.

[Mo66] Moore, R. E., Interval Analysis. Prentice-Hall, Englewood Cliffs, New Jersey, 1966.

[Mo79] Moore, R. E., Methods and Applications of Interval Analysis. SIAM Studies in Applied Mathematics, Philadelphia, 1979.

[NJ01] Nedialko, N., Jackson, K., Pryce, J., *An Effective High-Order Interval Method for Validating Existence and Uniqueness of the Solution of an IVP for an ODE*, Reliable Computing **7** (2001), 449-465.

[Ov01] Overton, M. L., Numerical Computing with IEEE Floating Point Arithmetic. SIAM, Philadelphia, 2001.

[PrBi] PROFIL/BIAS – Programmer's Runtime Optimized Fast Interval Library/Basic Interval Arithmetic Subroutines. Available from `http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html`

[PC06] Pryce, J. D., Corliss, G. F., *Interval Arithmetic with Containment Sets*, Computing **78:3** (2006), 251–276.

[Ra96] Ratz, D., *Inclusion Isotone Extended Interval Arithmetic – A Toolbox Update*, FCINE Report No 5/1996, Universität Karlsruhe, 1996.

[Ru88] Rump, S. M., *Algorithms for Verified Inclusions: Theory and Practice*, in Reliability in Computing: The Role of Interval Methods in Scientific Computing, Academic Press, Boston, 1988.

[St95] Strichartz, R. S., The Way of Analysis. Jones & Bartlett, Boston, 1995.

[SK99] Schwarz, E. M., Krygowski, C. A., *The S/390 G5 floating-point unit*, IBM J. Res. Development **43** No. 5/6 (1999), 707-721.

[Su58] Sunaga, T., *Theory of an Interval Algebra and its Application to Numerical Analysis*. RAAG Memoirs, **2** (1958), 29-46.

[1] Forte Developer 7: C++ Interval Arithmetic Programming Reference. Available from
`http://docs.sun.com/app/docs/doc/816-2465`

[Tu02] Tucker, W., *A Rigorous ODE Solver and Smale's 14th Problem*. Found. Comp. Math., **2:1** (2002), 53-117.

[Wa56] Warmus, M., *Calculus of Approximations*. Bulletin de l'Academie Polonaise de Sciences, **4:5** (1956), 253-257.

[Wi63] Wilkinson, J. H., Rounding Errors in Algebraic Processes. Prentice-Hall, Englewood Cliffs, New Jersey, 1963

[Yo31] Young, R. C., *The algebra of multi-valued quantities*. Mathematische Annalen, **104** (1931), 260-290.