Chapter 1

Basic Arithmetic

In this section, we present an efficient algorithm due to Toom and Cook for multiplying two integers, which already considerably improves upon the method that most people have learned in school. We further investigate in methods for carrying out approximate computations on fixed-point and floating-point numbers, and we derive bounds on the occurring error when using approximate instead of exact arithmetic. In addition, we introduce the concepts of interval arithmetic and box-functions and show that these concepts yield a powerful and very practical approach for carrying out approximate arithmetic. This is due to the fact that adaptive bounds on the error can directly be computed "on the fly", and that these bounds are often much better than any a priori bounds obtained by a worst-case error analysis. Finally, we give an efficient method to compute an arbitrary good approximation of the quotient of two integers or, more generally, two arbitrary complex values.

1.1 The School Method for Integer Multiplication

We represent integers $a \in \mathbb{Z}$ as digit strings with respect to a fixed base $B \in \mathbb{N}_{\geq 2}$. That is,

$$a = (-1)^s \cdot \sum_{i=0}^{n-1} a_i \cdot B^i, \text{ with } s \in \{0,1\} \text{ and } a_i \in \{0,\ldots,B-1\} \text{ for all } i = 0,\ldots,n-1.$$

We call the a_i's the *digits* and s the *sign digit* of a with respect to B. For convenience, we also write (if B is fixed)

$$a = (-1)^s \; a_{n-1}a_{n-2}\ldots a_0$$

if the base B is fixed.

Example: Important bases are $B = 2, 10, 16$, and 2^k for some $k \in \mathbb{N}$. The integer 29 writes as

$$29 = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 = 11101.$$

The *length* (or *bitsize* for $B = 2$) of an integer a with respect to B is defined as the number of digits needed to represent a. For convenience, we use the term *n-digit number* to denote an integer of length n. Notice that any n-digit number can always be considered as an N-digit number for arbitrary $N \geq n$. This is advantageous in the analysis of many algorithms as it allows us to assume that the length of the input is a power of 2 (or some other value k).

Algorithm 1: School Method for Addition

Input : Two non-negative n-digit integers $a = a_{n-1} \ldots a_0$ and $b = b_{n-1} \ldots b_0$.
Output: An $(n+1)$-digit integer $c = c_n \ldots c_0$ with $c = a + b$.

```
1 γ_0 := 0
2 for i = 0, ..., n − 1 do
3 |   Recursively define
4 |   γ_{i+1} · B + c_i = a_i + b_i + γ_i with c_i, γ_i ∈ {0, ..., B − 1}
5 c_n := γ_n
6 return c_n ... c_0
```

We mainly consider two different ways of measuring the efficiency of an algorithm. The first one is to count the number of additions and multiplications between integers that an algorithm needs to return a result. This is referred to as the *arithmetic complexity* of an algorithm. Notice that the arithmetic complexity might be unrelated to the actual running time of an algorithm as the involved integers can be arbitrarily large. Hence, a more meaningful and precise way of measuring the efficiency of an algorithm is to count instead the number of *primitive operations* (or bit operations if the base B equals 2) that are carried out by the algorithm, often referred to as the *bit complexity* of an algorithm. Notice that the result of a primitive operations is always a one- or two-digit number.

Example: A prominent example is Gaussian elimination for solving a linear system in n unknowns. It is easy to see that the method uses $O(n^3)$ arithmetic operations, hence the arithmetic complexity of Gaussian elimination is polynomial in the input size. However, a straight forward analysis does NOT guarantee that the intermediate results as computed by the algorithm (which are rationals if the input matrix has integer entries) have size that is polynomial in the size of the input, thus it is not obvious that Gaussian elimination actually constitutes a polynomial time algorithm for solving linear systems. A more refined argument however shows that by recursively removing common factors of the intermediate results, it can be guaranteed that all intermediate results have polynomial size. We will go into more detail in one of the exercises. Later, we will also consider a different approach based on modular computation that does not come with any of these drawbacks.

We now review and analyze the school method for adding and multiplying two non-negative n-digit integers $a = a_{n-1} \ldots a_0$ and $b = b_{n-1} \ldots b_0$. We first start with addition; see Algorithm 1. The γ_i's are called *carries*. Using induction, it is easy to see that $\gamma_i \in \{0, 1\}$ for all i. Further notice that γ_{i+1} is non-zero if and only if the sum of the two digits a_i and b_i and the previous carry γ_i is larger than the base B. We also remark that, for subtraction (i.e. the computation of $a - b$), we can assume that $a \geq b$. The recursion for c_i and γ is then almost identical. More specifically, we have

$$-\gamma_{i+1} \cdot B + c_i = a_i - b_i - \gamma_i \text{ with } c_i, \gamma_i \in \{0, \ldots, B-1\}.$$

The proof of the following theorem is straight-forward.

Theorem 1.1.1. *The school method for adding (or subtracting) two $n-$digit numbers requires at most $2n$ primitive operations. The addition of an m-digit number and an n-digit number uses at most $m + n + 2$ primitive operations.*

Algorithm 2: School Method for Multiplication

Input : Two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$.
Output: A $2n$-digit integer $c = c_{2n-1} \dots c_0$ with $c = a \cdot b$.

```
1  P_0 := 0
2  for j = 0, ..., n-1 do
3      for i = 0, ..., n-1 do
4          Define
5          a_i · b_j = c_ij · B + d_ij with c_ij, d_ij ∈ {0, ..., B-1}
6      c_j := c_{n-1,j} ... c_{0,j}0
7      d_j := d_{n-1,j} ..., d_{0,j}
8      p_j = p_{n,j} ... p_{0,j} := c_j + d_j
       // * Notice that p_j = a · b_j, and thus a · b = Σ_{j=0}^{n-1} p_j · B^j *//
9      P_{j+1} := P_j + p_j · B
10 return P_n
```

In the next step, we consider the school method for multiplying integers; see Algorithm 2. Let us count the number of primitive operations that Algorithm 2 needs:

- The computation of each product $a_i \cdot b_j$ requires one primitive operations, thus n^2 many primitive operations in total.
- Computing each of the integers p_j amounts for adding two $(n+1)$-digit numbers. Hence, in total, we need $2n(n+1)$ primitive operations.
- For computing P_n we need n additions each involving $2n$-digit numbers. Thus, we need $2n^2$ many primitive operations for this step.

We now obtain the following result. For the second claim on the complexity of computing the product of an m-digit number and an n-digit number, a completely analogous argument applies.

Theorem 1.1.2. *Using the school method, we need at most $5n^2 + 2n = O(n^2)$ primitive operations to multiply two n-digit numbers. Multiplication of an n-digit number and an m-digit number needs $O(mn)$ primitive operations.*

Exercise 1.1.3. *Let $f = a_0 + \dots + a_d \cdot x^d \in \mathbb{Z}[x]$ be a polynomial of degree d with integer coefficients of length at most L, and let $m \in \mathbb{Z}$ be an ℓ-digit number. Show that*

(a) $f(m)$ is a $O(d\ell + L)$-digit number.

(b) Computing $f(m)$ using Horner's method

$$f(m) = a_0 + m \cdot (a_1 + m \cdot (a_2 + \cdots m \cdot (a_{d-1} + m \cdot a_d))$$

and the school method for multiplication uses $O(d \cdot (d\ell^2 + \ell \cdot L)))$ primitive operations.

We will later see that it is even possible to compute $f(m)$ in only $\tilde{O}(d \cdot (\ell + L))$ primitive operations, where $\tilde{O}(.)$ means that poly-logarithmic factors are suppressed, that is, $\tilde{O}(T) = O(T \cdot (\log T)^c)$ for some constant c.

Exercise 1.1.4. *Let $A = (a_{i,j})_{i,j=1,\dots,n} \in \mathbb{Z}^{n\times n}$ be an $n \times n$-matrix with integer entries $a_{i,j}$ of length at most L.*

(a) Derive an upper bound on the number of primitive operations that are needed to compute the inverse A^{-1} of A.

(b) Show that the entries of A^{-1} are rational numbers with numerators and denominators of length $O(n(L + \log n))$.

(c) Suppose that Gaussian elimination with pivoting is used to compute the determinant of A. Further suppose that, after each iteration, we reduce all intermediate entries $a'_{i,j} = \frac{p}{q} \in \mathbb{Q}$, that is, we ensure that $\gcd(p, q) = 1$. Show that p and q can be represented using $O(n^2(L+\log n))$ digits and conclude that Gaussian elimination constitutes a polynomial time algorithm for computing determinants.*

Hints: For (a), consider Gaussian elimination to compute A^{-1} and derive a bound on the numerators and denominators of the rational entries of the matrices produced after each iteration. For (b), use Cramer's Rule to write the entries of A^{-1} as fractions of determinants of suitable $n \times n$-matrices and use the definition of the determinant to bound the size of the numerator and denominator. For (c), show that, in each iteration, the pivot element can be written as the quotient of the determinants of two sub-matrices of A.

1.2 The Toom-Cook Algorithm

We now investigate in algorithms for multiplying integers that are considerably faster than the school method. We start with a simple algorithm due to Karatsuba (from 1960). Its running time $O(n^{\log_2 3})$ already constitutes a considerable improvement upon the running time $O(n^2)$ of the school method. Then, we show how to generalize the approach to achieve a running time $O(n^{1+\epsilon})$ for arbitrary $\epsilon > 0$.

Let $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$ be integers of length n. We first write

$$a = a_{n-1} \dots a_0 = a' \cdot B^{\lceil n/2 \rceil} + a'', \text{ and}$$
$$b = b_{n-1} \dots b_0 = b' \cdot B^{\lceil n/2 \rceil} + b'',$$

with integers a', a'', b', b'' of length $\lceil n/2 \rceil$. Then, it holds that

$$\begin{aligned} a \cdot b &= (a' \cdot B^{\lceil n/2 \rceil} + a'') \cdot (b' \cdot B^{\lceil n/2 \rceil} + b'') \\ &= a'b' \cdot B^{2\lceil n/2 \rceil} + (a'b'' + a'' \cdot b') \cdot B^{\lceil n/2 \rceil} + a''b'' \\ &= \underbrace{a'b'}_{=:P_1} \cdot B^{2\lceil n/2 \rceil} + [\underbrace{(a' + a'')(b' + b'')}_{=:P_2} - (a'b' + a''b'')] \cdot B^{\lceil n/2 \rceil} + \underbrace{a''b''}_{=:P_3} \end{aligned} \tag{1.1}$$

What have we gained in the last step? They crucial point is that, when passing from the second line to the last line, we reduced the problem to three (instead of four!) multiplications and six (instead of three) additions. Notice that there are actually five multiplications, however, each of the products P_1 and P_2 appears twice, and thus only 3 different products need to be computed. So the total number of additions and multiplication has increased, however, additions are much cheaper than multiplications. We can now recursively use the

Algorithm 3: Karatsuba Multiplication (1960)

Input : Two non-negative n-digit integers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$.
Output: A $2n$-digit integer $c = c_{2n-1} \dots c_0$ with $c = a \cdot b$.

1 **if** $n \leq 4$ **then**
2   Compute $c = a \cdot b$ using Algorithm 2
3 **else**
4   Define

$$a = a_{n-1} \dots a_0 = a' \cdot B^{\lceil n/2 \rceil} + a'', \text{ and}$$
$$b = b_{n-1} \dots b_0 = b' \cdot B^{\lceil n/2 \rceil} + b'',$$

    with integers a', a'', b', b'' of length $n/2$.
5     $A := a' + a''$
6     $B := b' + b''$
7   Compute $P_1 := a' \cdot b'$, $P_2 := A \cdot B$, and $P_3 := a'' \cdot b''$ by recursively calling Algorithm 3.
8     $P := P_1 \cdot B^{2\lceil n/2 \rceil} + (P_2 - P_1 - P_3) \cdot B^{\lceil n/2 \rceil} + P_3$
9 **return** P

above approach for multiplication until all remaining multiplications are numbers with four or less digits; see Algorithm 3

Theorem 1.2.1. *Using Karatsuba multiplication, we need $O(n^{\log 3}) = O(n^{1.58\dots})$ primitive operations to multiply two n-digit numbers.*

Proof. Let $T(n)$ denote the maximal number of operations needed to multiply two n-digit numbers using the Karatsuba algorithm. If $n \leq 4$, Theorem 1.1.2 yields that $T(n) \leq 5n^2 + 2n \leq 88$. For $n \geq 5$, it holds that

$$T(n) \leq 3 \cdot T(\lceil n/2 \rceil + 1) + 6 \cdot (4n).$$

as we need to compute 3 products involving $\lceil n/2 \rceil$- or $(\lceil n/2 \rceil + 1)$-digit numbers and 6 additions involving $2n$-digit numbers. Now, a general version of the Master Theorem (e.g. see [MS08, Sec. 2.6]) yields a total running time of size $O(n^{\log_2 3})$. □

Remark. For readers who are not familiar with the general Master Theorem, we give the following direct argument from [MS08], which also yields an explicit bound for $T(n)$. For $\ell \in \mathbb{N}_{\geq 1}$, we first prove that

$$T(2^\ell + 2) \leq 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2)$$

using induction on ℓ. For $\ell = 1$, the claim is obviously true as $T(4) \leq 88$. For $\ell \geq 2$, we thus conclude from the induction hypothesis and the above recursive formula for $T(n)$ that

$$\begin{aligned} T(2^\ell + 2) &\leq 3 \cdot T(2^\ell + 2) + 12 \cdot (2^\ell + 2) \\ &\leq 3 \cdot [33 \cdot 3^{\ell-1} + 12 \cdot (2^\ell + 2(\ell - 1) - 2)] + 12 \cdot (2^\ell + 2) \\ &= 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2). \end{aligned}$$

Notice that our special choice for n (i.e. $n = 2^\ell + 2$) guarantees that $\lceil n/2 \rceil + 1 = 2^{\ell-1} + 2$ is again of the same form, and thus we can recursively apply the induction hypothesis on $T(\lceil n/2 \rceil + 1)$. It remains to derive a bound on $T(n)$ for arbitrary n. Setting $\ell := \lceil \log n \rceil \leq 1 + \log n$, we have

$$\begin{aligned} T(n) \leq T(2^\ell) &\leq 33 \cdot 3^\ell + 12 \cdot (2^{\ell+1} + 2\ell - 2) \\ &\leq 33 \cdot 3 \cdot 3^{\log n} + 12 \cdot (4 \cdot 3^{\log n} + 2(1 + \log n) - 2) \\ &\leq 99 \cdot n^{\log 3} + 48 \cdot \cdot n + 24 \cdot \log n. \end{aligned}$$

We now consider the following approach due to Toom and Cook (1966), which extends Karatsuba's idea; see Algorithm 4. The first step is similar as in Karatsuba's method, however, instead of splitting each of the input numbers into two almost equally sized parts, we now consider a split into k parts, where $k \in \mathbb{N}_{\geq 2}$ is an arbitrary but fixed constant. That is, with $m := \lceil n/k \rceil$, we write

$$\begin{aligned} a &= a^{(0)} + a^{(1)} \cdot B^m + \cdots + a^{(k-1)} \cdot B^{(k-1)\cdot m}, \text{ and} \\ b &= b^{(0)} + b^{(1)} \cdot B^m + \cdots + b^{(k-1)} \cdot B^{(k-1)\cdot m}, \end{aligned}$$

such that each integer $a^{(i)}$ and $b^{(i)}$ has length at most m. Now, let $f(x) := \sum_{i=0}^{k-1} a^{(i)} \cdot x^i$ and $g(x) := \sum_{i=0}^{k-1} b^{(i)} \cdot x^i$ be corresponding polynomials of degree $k-1$ with coefficients $a^{(i)}$ and $b^{(i)}$. Then, it holds that $a \cdot b = f(B^m) \cdot g(B^m) = h(B^m)$, where

$$h(x) = \sum_{i=0}^{2k-2} c^{(i)} \cdot x^i := f(x) \cdot g(x).$$

Notice that the coefficients $c^{(i)}$ of h are integers of length at most $O(m)$. Now, suppose that we know these coefficients, then we can easily compute $a \cdot b$ by shifting each of the coefficients $c^{(i)}$ by $i \cdot m$ digits and adding up the resulting integers. The cost for these additions (there are only constantly many!) is then bounded by $O(n)$. Hence, we have reduced the problem of computing the product $a \cdot b$ of two integers of length n to the problem of computing a product $g(x) \cdot h(x)$ of polynomials of degree less than k and with coefficients of length at most $\lceil n/k \rceil$. For the latter problem, we consider an *evaluation/interpolation approach*, that is, we first evaluate f and g at $2k-1$ many different points $x_0, \ldots, x_{2k-2} \in \mathbb{Z}$ of constant length. Typically, we consider $x_j := j$ for $j = 0, \ldots, 2k-2$ but also other choices are possible. Then, the resulting integer values $f_j = f(x_j)$ and $g_j := g(x_j)$ are of length $O(m)$ according to Exercise 1.1.3. For computing the k products $h_j := f_j \cdot g_j = f(x_j) \cdot g(x_j) = h(x_j)$, we call the multiplication algorithm recursively. In the third step, we interpolate $h(x)$ from its values h_j at the points x_j. Notice that

$$\underbrace{\begin{pmatrix} 1 & x_0 & \cdots & x_0^{2k-2} \\ 1 & x_1 & \cdots & x_1^{2k-2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{2k-2} & \cdots & x_{2k-2}^{2k-2} \end{pmatrix}}_{=:V} \cdot \begin{pmatrix} c^{(0)} \\ c^{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = \begin{pmatrix} h(x_0) \\ h(x_1) \\ \vdots \\ h(x_{2k-2}) \end{pmatrix} = \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix},$$

Algorithm 4: Toom-Cook-k Algorithm

Input : Two non-negative integers a and b of length at most n.
Output: The product $c = a \cdot b$.

1 Write

$$a = a^{(0)} + a^{(1)} \cdot B^m + \cdots + a^{(k-1)} \cdot B^{(k-1)\cdot m}, \text{ and}$$
$$b = b^{(0)} + b^{(1)} \cdot B^m + \cdots + b^{(k-1)} \cdot B^{(k-1)\cdot m},$$

with $m := \lceil n/k \rceil$ and integers $a^{(i)}$, $b^{(i)}$ of length at most m.

2 $f(x) := a^{(0)} + a^{(1)} \cdot x + \cdots + a^{(k-1)} \cdot x^{k-1}$

3 $g(x) := b^{(0)} + b^{(1)} \cdot x + \cdots + b^{(k-1)} \cdot x^{k-1}$

4 **for** $j = 0, \ldots, 2k-2$ **do**

5 Define $x_j = j$

*// * We can also choose other values for x_j unless the x_j's are pairwise distinct and of constant length *//*

6 Compute $f_j := f(x_j)$ and $g_j := g(x_j)$

7 Compute $h_j := f_j \cdot g_j$ by calling the Algorithm 4 recursively.

8 Compute the inverse V^{-1} of the *Vandermonde Matrix*

$$V := \mathrm{Vand}(x_0, \ldots, x_{2k-2}) := \begin{pmatrix} 1 & x_0 & \cdots & x_0^{2k-2} \\ 1 & x_1 & \cdots & x_1^{2k-2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{2k-2} & \cdots & x_{2k-2}^{2k-2} \end{pmatrix}$$

Compute

$$\begin{pmatrix} c^{(0)} \\ c_{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = V^{-1} \cdot \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix}$$

$C_0 = c^{(0)}$

9 **for** $j = 1, \ldots, 2k-2$ **do**

10 $C_j = C_j + B^{mj} \cdot c^{(j)}$

11 **return** $C_{2k-2} = c = a \cdot b$

where $V = \mathrm{Vand}(x_0, \ldots, x_{2k-2})$ is the so-called *Vandermonde-Matrix* of $x_0, \ldots, x_{2k-2}$. Hence, we can compute the coefficients $c^{(i)}$ of $h(x)$ from its values h_j at the $2k-1$ points x_j as

$$\begin{pmatrix} c^{(0)} \\ c^{(1)} \\ \vdots \\ c^{(2k-2)} \end{pmatrix} = V^{-1} \cdot \begin{pmatrix} h_0 \\ h_1 \\ \vdots \\ h_{2k-2} \end{pmatrix}$$

Since k is a constant and since each entry of V is of constant size, only a constant number

of primitive operations is needed to compute V^{-1}. Computing the product of V^{-1} and the vector $(h_0, \dots, h_{2k-2})^t$ needs $O(n)$ primitive operations as each h_j has length $O(n)$. Finally, we compute $c = a \cdot b$ as the sum of the $2k-1$ integers $c_j \cdot B^j$, for $j = 0, \dots, 2k-2$, which also uses $O(n)$ primitive operations.

In summary, we thus obtain the following recursion for the computation time $T(n)$ of the Toom-Cook-k Algorithm:

$$T(n) \leq (2k-1) \cdot T(\lceil n/k \rceil) + O(n).$$

Again, the Master Theorem yields the following result:

Theorem 1.2.2. *For a fixed integer $k \in \mathbb{N}_{\geq 2}$, the Toom-Cook-$k$ Algorithm uses $O(n^{\frac{\log(2k-1)}{\log k}})$ primitive operations to multiply two n-digit numbers.*

From the above theorem and the fact that $\lim_{k \mapsto \infty} \frac{\log(2k-1)}{\log k} = 1$, we conclude that, for any fixed $\epsilon > 0$, there exists an algorithm with running time $O(n^{1+\epsilon})$ to multiply to n-digit numbers. In the next chapter, will discuss a method due to Schönhage and Strassen (1971) that even yields a running time of size $O(n \cdot \log^c(n))$, with some constant $c > 1$. The method is similar to the Toom-Cook approach in the sense that it considers the input integers as polynomials and then computes the product of the polynomials using an evaluation/interpolation-approach. The main difference however is that n-digits numbers are considered as polynomials of degree $n-1$ (and not k for some fixed constant k) and that the interpolation points are chosen to be the $2n$-th roots of unity. Here, the crucial point is that evaluating and interpolating a polynomial at the roots of unity can be done in a very efficient way.

Exercise 1.2.3. *Show that Karatsuba's method can be considered as a special case of Toom-Cook-2. For this, you need to choose suitable interpolation points x_0, x_1, x_2 in the Toom-Cook-2 algorithm.*

Hint: You may choose $x_0 = \infty$ as one of the interpolation points, where we define $P(\infty) := P_d$ for a polynomial $P(x) = P_0 + \cdots + P_d \cdot x^d$. For the interpolation step, you cannot use the Vandermonde matrix any more but need a more direct approach instead.

Exercise 1.2.4. *For two integers $a = a^{(0)} + a^{(1)} \cdot B^{\lceil n \rceil} + a^{(3)} \cdot B^{2\lceil n \rceil}$ and $b = b^{(0)} + b^{(1)} \cdot B^{\lceil n \rceil} + b^{(3)} \cdot B^{2\lceil n \rceil}$ of length n, use the Toom-Cook-3 approach to derive a relation between the values $a^{(i)}$ and $b^{(i)}$ that is similar to the relation in (1.1) as considered in Karatsuba's method.*

1.3 Approximate Computation

1.3.1 Fixed Point Arithmetic

A common approach when dealing with non-integer values a (e.g. $1/3$, $\sqrt{2}$, or π) is to approximate them by rational numbers $\tilde{a} = m \cdot B^{-\rho}$, with B the working base, $m \in \mathbb{Z}$ and $\rho \in \mathbb{N}$, such that $|a - \tilde{a}| \leq B^{-\rho+1}$. That is, $\tilde{a}$ constitutes the best approximation of a among all *fixed-point number* with base B and precision ρ:

$$\mathbb{F}_{B,\rho} := \{a = (-1)^s \cdot B^{-\rho} \cdot \sum_{i=0}^{n-1} a_i B^i \text{ with } n \in \mathbb{N},\ s \in \{0,1\},\ \text{and } a_i \in \{0, \dots, B-1\}\}$$

If B and ρ are clear from the context, we also write $\mathbb{F} = \mathbb{F}_{B,\rho}$. For convenience, we also write

$$a = (-1)^s \; a_{n-1} \ldots a_{\rho+1} a_\rho, a_{\rho-1} \ldots a_0$$

for an arbitrary element $a = (-1)^s \cdot B^{-\rho} \cdot \sum_{i=0}^{n-1} a_i B^i \in \mathbb{F}_{B,\rho}$. The *length of* a (with respect to B) is defined as the number n of digits that is needed to represent a. It is common to consider the base $B = 2$ and to work with so called *dyadic* numbers (also called dyadic rationals). These are exactly the fixed point numbers with respect to base 2 and arbitrary but finite precision:

$$\mathbb{D} := \bigcup_{\rho=0}^{\infty} \mathbb{F}_{2,\rho} = \{p \cdot 2^{-\rho} : p \in \mathbb{Z} \text{ and } \rho \in \mathbb{N}\}.$$

In what follows, we always assume that the base B and the precision ρ is fixed. For an arbitrary real value x, we define

$$\mathrm{flu}(x) := \min\{a \in \mathbb{F} : x \leq a\}$$

and

$$\mathrm{fld}(x) := \max\{a \in \mathbb{F} : x \geq a\}.$$

the two *rounding functions* to the nearest fixed-point number that is larger/smaller than or equal to a. $\mathrm{fl}(.)$ defines the *rounding to nearest*, that is, $\mathrm{fl}(x) = \mathrm{flu}(x)$ if $|\,\mathrm{flu}(x) - x| < |\,\mathrm{fld}(x) - x|$ and $\mathrm{fl}(x) = \mathrm{fld}(x)$ if $|\,\mathrm{fld}(x) - x| < |\,\mathrm{flu}(x) - x|$. In case of ties (i.e. $|\,\mathrm{fld}(x) - x| = |\,\mathrm{flu}(x) - x|$), we round to even, that is, $\mathrm{fl}(x) = \mathrm{flu}(x)$ if the last digit of $\mathrm{flu}(x)$ is even, otherwise $\mathrm{fl}(x) = \mathrm{fld}(x)$. For each arithmetic operations $\circ \in \{+, -, \cdot\}$, we now consider a corresponding approximate variant $\tilde{\circ}$, where we use $\mathrm{fl}(.)$ to round the exact result to a nearby number in $\mathbb{F}$:

Definition 1.3.1. *For $x, y \in \mathbb{R}$ and $\circ \in \{+, -, \cdot\}$, we define*

$$x \tilde{\circ} y := \mathrm{fl}(\mathrm{fl}(x) \circ \mathrm{fl}(y)).$$

In particular, we have $x \tilde{\circ} y := \mathrm{fl}(x \circ y)$ for $x, y \in \mathbb{F}$.

Notice that the above definition yields a canonical way of approximately evaluating a polynomial $f(x) = a_0 + \cdots a_d \cdot x^d \in \mathbb{R}[x]$ at an arbitrary real value x. More precisely, we consider some evaluation method (e.g. Horner Evaluation) and replace each of the occurring arithmetic operations $\circ$ by the corresponding fixed point variant $\tilde{\circ}$. We denote the so-obtained result by $f_{\mathbb{F}}(x)$. We remark at this point that the result may crucially depend on the chosen evaluation method. That is, we might get completely different values when using Horner Evaluation instead of the "classical" way of evaluating the polynomial, that is, by first computing all powers x^i of x, then multiplying each power with the corresponding coefficient a_i, and finally summing up the obtained values. In other terms, it does not necessarily hold that

$$a_0 \tilde{+} x \tilde{\cdot} (a_1 \tilde{+} \cdots (a_{d-1} \tilde{+} x \tilde{\cdot} a_d) \ldots) = a_0 \tilde{+} a_1 \tilde{\cdot} x \tilde{+} \cdots \tilde{+} a_d \tilde{\cdot} x \tilde{\cdot} x \cdots x \tilde{\cdot} x$$

Exercise 1.3.2. *Give an example where Horner Evaluation and classical evaluation give different results for $f_{\mathbb{F}}(x)$.*

The above approach for approximately evaluating a univariate polynomial at a point then further extends to polynomials $F(\mathbf{x}) \in \mathbb{R}[\mathbf{x}] = \mathbb{R}[x_1, \ldots, x_n]$ in several variables. Since each complex number z can be written as $z = x + \mathbf{i} \cdot y$ with $x, y \in \mathbb{R}$, and since each addition and

multiplication in $\mathbb{C}$ amounts for a constant number of additions and multiplications in $\mathbb{R}$, we may further extend the approach to polynomials with complex coefficients. In this case, the set of *complex fixed point numbers* is given as

$$\mathbb{F}_{\mathbb{C}} := \mathbb{F} + \mathbf{i} \cdot \mathbb{F},$$

and the set of *complex dyadic numbers* is given as

$$\mathbb{D}_{\mathbb{C}} := \mathbb{D} + \mathbf{i} \cdot \mathbb{D}.$$

In the next step, we investigate the error when performing a series of additions and multiplications using fixed point arithmetic. Assume that we are given approximations $\tilde{x}, \tilde{y} \in \mathbb{F}_{\mathbb{C}}$ of two complex numbers $x, y \in \mathbb{C}$ with $|x - \tilde{x}| < \epsilon_x$ and $|x - \tilde{x}| < \epsilon_y$. Then, it holds that

$$|(\tilde{x} \tilde{+} \tilde{y}) - (x + y)| \leq \sqrt{2} \cdot B^{-(\rho+1)} + |(\tilde{x} + \tilde{y}) - (x + y)| < B^{-\rho} + \epsilon_x + \epsilon_y, \tag{1.2}$$

and the same error bound holds true for subtraction. For multiplication, we have

$$|\tilde{x} \tilde{\cdot} \tilde{y} - x \cdot y| \leq \sqrt{2} \cdot B^{-(\rho+1)} + |(\tilde{x} \cdot \tilde{y}) - x \cdot y| < B^{-\rho} + \epsilon_x \cdot |y| + \epsilon_y \cdot |x| + \epsilon_x \cdot \epsilon_y. \tag{1.3}$$

From the above error bounds, we can now derive a bound on the error $|f(x_0) - f_{\mathbb{F}}(x_0)|$ that we obtain when using Horner evaluation and fixed point arithmetic to compute the value of a polynomial f at a complex point x_0.

Theorem 1.3.3. *For any $x_0 \in \mathbb{C}$ and any polynomial $f \in \mathbb{C}[x]$ of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\geq 0}$, it holds that*

$$|f(x_0) - f_{\mathbb{F}}(x_0)| < 4(d+1)^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^d.$$

if Horner Evaluation and fixed point arithmetic with a precision $\rho \geq \log d$ is used for the evaluation of f at x_0.

Proof. We argue by induction on the degree d of $f = a_0 + \cdots + a_d \cdot x^d$. Obviously, the error bound is true for $d = 0$ as

$$|a_0 - \mathrm{fl}(a_0)| \leq \sqrt{2} \cdot B^{-(\rho+1)},$$

When using Horner evaluation to evaluate a polynomial f of degree $d \geq 1$ at x_0, we first evaluate $\hat{f} := a_1 + a_2 \cdot x + \cdots + a_d \cdot x^{d-1}$ at x_0, then multiply the result by x_0 and eventually add a_0. Using fixed point arithmetic with precision ρ, our induction hypotheses yields that

$$|\hat{f}_{\mathbb{F}}(x_0) - \hat{f}(x_0)| < \epsilon := 4d^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^{d-1}$$

Since $|\hat{f}(x_0)| \leq d \cdot 2^L \cdot \max(1, |x_0|)^{d-1}$ and $|x_0 - \mathrm{fl}(x_0)| \leq \sqrt{2} \cdot B^{-(\rho+1)} < B^{-\rho}$, we conclude from (1.3) that

$$\begin{aligned}
|x_0 \cdot \hat{f}(x_0) - \mathrm{fl}(x_0) \tilde{\cdot} \hat{f}_{\mathbb{F}}(x_0))| &< \sqrt{2} \cdot B^{-(\rho+1)} + \epsilon \cdot |x_0| + B^{-\rho} \cdot |\hat{f}(x_0)| + B^{-\rho} \cdot \epsilon \\
&< B^{-\rho} + \epsilon \cdot \max(1, |x_0|) + B^{-\rho} \cdot |\hat{f}(x_0)| + \frac{\epsilon \cdot \max(1, |x_0|)}{d} \\
&< B^{-\rho} \cdot [1 + 5d \cdot 2^L \cdot \max(1, |x_0|)^{d-1} + 4d^2 \cdot 2^L \cdot \max(1, |x_0|)^d]. \\
&\leq B^{-\rho} \cdot \max(1, |x_0|)^d \cdot 2^L \cdot (1 + 5d + 4d^2)
\end{aligned}$$

Adding the constant a_0 increases the error by less than $2 \cdot B^{-\rho}$ due to (1.2). Hence, the total error is bounded by

$$B^{-\rho} \cdot 2^L \cdot (3 + 5d + 4d^2) \cdot \max(1, |x_0|)^d \leq 4(d+1)^2 \cdot 2^L \cdot B^{-\rho} \cdot \max(1, |x_0|)^d.$$

Hence, the claim follows. □

1.3.2 Interval Arithmetic

Instead of computing an approximation of the value $f(x_0)$ that a function $f : \mathbb{R} \mapsto \mathbb{R}$ (or more general, $f : \mathbb{C} \mapsto \mathbb{C}$) takes at a specific point $x_0 \in \mathbb{R}$ (or $x_0 \in \mathbb{C}$), it is often useful to compute an approximation of the image $f([a,b])$ (or $f([a,b] + \mathbf{i} \cdot [c,d])$) of an interval $[a,b]$ (rectangle $[a,b] + \mathbf{i} \cdot [c,d]$) under the mapping f.

Definition 1.3.4 (Interval Extensions and Box Functions)**.** *Let $f : \mathbb{R} \mapsto \mathbb{R}$ be an arbitrary function. An* interval extension *$\Box f : H \mapsto H$ of f is a function from the halfplane $H := \{[a,b] : a,b \in \mathbb{R}$ with $a \leq b\}$ of intervals $X = [a,b]$ to itself such that $f(x) \in \Box f(X)$ for all $x \in X$. For continuous f, $\Box f$ is a* continuous interval extension *(or* box-function*) if*

$$\bigcap_{i=1}^{\infty} \Box f(X_i) = f(x_0)$$

for any sequence $X_1 \supset X_2 \supset \cdots$ such that $\bigcap_{i=1}^{\infty} X_i$ contains only a single point x_0.

In simpler terms, an interval extension $\Box f$ of f is a function that maps an interval $[a,b]$ to an interval $[A,B]$ such that $f(x) \in [A,B]$ for any $x \in [a,b]$. Notice that this is not a very restricting condition as we can simply choose $\Box f$ as the function that maps any interval to $(-\infty, +\infty)$. However, for a box function, it must also hold that $[A,B]$ shrinks to one point $(f(x_0))$ if $[a,b]$ shrinks to one point (x_0).

We further remark that Definition 1.3.4 further generalizes to complex valued functions $f : \mathbb{C} \mapsto \mathbb{C}$. Then, an interval extension $\Box f : H_{\mathbb{C}} \mapsto H_{\mathbb{C}}$ computes for each rectangle $R = [a,b] + \mathbf{i} \cdot [c,d] \in H_{\mathbb{C}} := H + \mathbf{i} \cdot H$ a rectangle $\Box f(B) \in H_{\mathbb{C}}$ with $f(B) \subset \Box f(B)$. The definition of a box function is also completely analogous to the real case. We now show how to compute a box-function for a polynomial. For this, we introduce the concept of interval-arithmetic.

Definition 1.3.5 (Interval Arithmetic)**.** *Let $[a,b]$ and $[c,d]$ be arbitrary intervals and λ a non-negative real number. Then, we define*

$$\begin{aligned}
\lambda \cdot [a,b] &:= [\lambda \cdot a, \lambda \cdot b] \\
-[a,b] &:= [-b, -a] \\
[a,b] \boxplus [c,d] &:= [a+c, b+d] \\
[a,b] \boxminus [c,d] &:= [a,b] \boxplus [-c,-d], \text{ and} \\
[a,b] \boxdot [c,d] &:= [\min(ab, bd, ad, bc), \max(ab, bd, ad, bc)]
\end{aligned}$$

The above rules then extend to arithmetic operations on rectangles in $\mathbb{C}$ in a straight forward way. In particular, for $R = [a,b] + \mathbf{i} \cdot [c,d]$ and $R' := [a',b'] + \mathbf{i} \cdot [c',d']$, we have

$$\begin{aligned}
R \boxplus R' &:= [a,b] \boxplus [a',b'] + \mathbf{i} \cdot ([c,d] \boxplus [c',d']), \\
R \boxdot R' &:= [a,b] \boxdot [a',b'] \boxminus [c,d] \boxdot [c',d'] + \mathbf{i} \cdot ([a,b] \boxdot [c',d'] \boxplus [a',b'] \boxdot [c,d]).
\end{aligned}$$

Often, we have to restrict to fixed point arithmetic instead of exact arithmetic. Similar to the definition of fl(.), which rounds a real (or complex) value to its best approximation in $\mathbb{F}$ (or $\mathbb{F}_{\mathbb{C}}$), we introduce the following rounding function for intervals (rectangles in $\mathbb{C}$):

$$\mathrm{Fl} : H_{\mathbb{C}} \mapsto H_{\mathbb{C}} : \mathrm{Fl}([a,b] + \mathbf{i} \cdot [c,d]) := [\mathrm{fld}(a), \mathrm{flu}(b)] + \mathbf{i} \cdot [\mathrm{fld}(c), \mathrm{flu}(d)]$$

Hence, Fl(.) rounds each of the vertices of a rectangle B to the nearest corresponding approximations in $\mathbb{F}_{\mathbb{C}}$ such that $\mathrm{Fl}(B)$ contains B. We can now define arithmetic operations on intervals (rectangles) using fixed point arithmetic.

Definition 1.3.6 (Fixed Point Interval Arithmetic)**.** *Let $[a,b]$ and $[c,d]$ be arbitrary intervals with $a,b,c,d \in \mathbb{F}$ and $\lambda \in \mathbb{R}$ a non-negative real number. Then, we define*

$$\begin{aligned}
[a,b]\tilde{\boxplus}[c,d] &:= \mathrm{Fl}([a,b] \boxplus [c,d]) \\
[a,b]\tilde{\boxminus}[c,d] &:= \mathrm{Fl}([a,b] \boxplus [-d,-c]), \\
[a,b]\tilde{\boxdot}[c,d] &:= \mathrm{Fl}([a,b] \boxdot [c,d]), \text{ and} \\
\lambda\tilde{\boxdot}[a,b] &:= [\mathrm{fld}(\lambda), \mathrm{flu}(\lambda)]\tilde{\boxdot}[a,b]
\end{aligned}$$

Again, the above rules for arithmetic operations on intervals extend in a straight forward manner to rectangles in $\mathbb{C}$. In addition, they induce interval extensions $\Box f$ and $\tilde{\Box} f$ for a polynomial $f \in \mathbb{R}[x]$. For this, we replace each arithmetic operation $\circ$ in the evaluation of f (e.g. when using Horner Evaluation) by the corresponding interval variant $\boxcirc$ and $\tilde{\boxcirc}$, respectively. Notice that $\Box f$ is a box-function, whereas this is not true for $\tilde{\Box} f$.

Exercise 1.3.7. *For any $x \in \mathbb{R}$ with $0 \le x \le 1$ and $k \in \mathbb{N}$, there exists a $\xi \in [0,x]$ such that*

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots + \frac{x^{4k}}{(4k)!} \cdot \cos(\xi) \quad \text{(Taylor Series Expansion with Remainder Term)}$$

Use the above formula to derive a box function $\Box\cos$ for $\cos$ for intervals $[a,b] \subset [0,1]$! Can you extend your approach to derive a box function for $\sin x$ and e^x.

We now investigate a bound on the size of the intervals (rectangles) that are obtained when performing a series of additions and multiplication according to the above rules. Notice that there are similarities to our considerations in the previous section, where we derived bounds on the error that occurs when adding or multiplying numbers using fixed point arithmetic. Namely, you might think of two rectangles $R := [a,b] + \mathbf{i} \cdot [c,d]$ and $R' := [a',b'] + \mathbf{i} \cdot [c',d']$ as approximations of its centers $m_R := \frac{a+b}{2} + \mathbf{i} \cdot \frac{c+d}{2}$ and $m_{R'} := \frac{a'+b'}{2} + \mathbf{i} \cdot \frac{c'+d'}{2}$ up to an error of size at most $\epsilon := \sqrt{2} \cdot w(R)$ and $\epsilon' := \sqrt{2} \cdot w(R')$, respectively, where $w(R) = \max(b-a, d-c)$ and $w(R') = \max(b'-a', d'-c')$ are defined as the *width* of R and R'. Then, the output of an arithmetic operation between R and R' can again be considered as an approximation of the corresponding arithmetic operation between m_R and $m_{R'}$. Hence, similarly to the bounds in (1.2) and (1.3), we obtain for any two rectangles R and R' with vertices in $\mathbb{F} + \mathbf{i} \cdot \mathbb{F}$ that

$$w(R \boxplus R') \le w(R \tilde{\boxplus} R') \le w(R) + w(R') + 2 \cdot B^{-\rho} \tag{1.4}$$

and

$$w(R \boxdot R') \le w(R) \cdot w(R') + |m_R| \cdot w(R') + |m_{R'}| \cdot w(R) + 2 \cdot B^{-\rho}. \tag{1.5}$$

Exercise 1.3.8. *Prove correctness of the inequalities in (1.4) and (1.5).*

Exercise 1.3.9. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\ge 0}$, let $\rho \in \mathbb{N}$ be a precision with $\rho > \log d$, and let $\mathbb{F}$ the corresponding set of fixed point numbers with precision ρ. Let $R = [a,b] + \mathbf{i} \cdot [c,d]$ be a rectangle of width*

$w(R) < \frac{1}{d}$ with vertices in $\mathbb{F} + \mathbf{i} \cdot \mathbb{F}$, and suppose that we compute $\Box f$ (and $\tilde{\Box} f$) using Horner Evaluation and fixed point interval arithmetic with a precision ρ. Then, it holds

$$w(\Box f(R)) \leq w(\tilde{\Box} f(R)) < 8 \cdot (d+1)^2 \cdot 2^L \cdot \max(1, |m_R|)^d \cdot w(R). \tag{1.6}$$

Hint: Consider a similar argument as in the proof of Theorem 1.3.3.

Notice that the bound (1.6) on $w(\Box f(R))$ and $w(\tilde{\Box} f(R))$ tends to zero if we consider a rectangle (square) R of width $c \cdot B^{-\rho}$, for some constant c, and the precision ρ tends to ∞. Hence, in order to compute an approximation of $f(x_0)$ for some complex value x_0, we may first approximate x_0 by some fixed point number $\tilde{x}_0 = \tilde{x}_{0,\Re} + \mathbf{i} \cdot \tilde{x}_{0,\Im} \in \mathbb{F}_{B,\rho} + \mathbf{i} \cdot \mathbb{F}_{B,\rho}$ such that $|x_0 - \tilde{x}_0| \leq B^{-\rho}$ and consider a rectangle

$$R := [\tilde{x}_{0,\Re} - B^{-\rho}, \tilde{x}_{0,\Re} + B^{-\rho}] + \mathbf{i} \cdot [\tilde{x}_{0,\Im} - B^{-\rho}, \tilde{x}_{0,\Im} + B^{-\rho}]$$

of width $2B^{-\rho}$ whose vertices are obtained by adding and subtracting $B^{-\rho}$ from the real and complex part of $\tilde{x}_0$. Then, R contains x_0 and we can use use interval arithmetic to compute the rectangle $\tilde{\Box} f(R)$, which contains $f(x_0)$. Its center m constitutes an approximation of $f(x_0)$ with $|m - f(x_0)| < w(\tilde{\Box} f(R))$. Hence, for computing an approximation m with $|m - f(x_0)| < \epsilon$, we can iteratively compute $\tilde{\Box} f(R)$ with increasing precision $\rho = 1, 2, 4, 8, \ldots$ until $w(\tilde{\Box} f(R)) < \epsilon$, and then return the center of $\tilde{\Box} f(R)$. Exercise 1.3.9 guarantees that we must succeed as soon as the precision ρ fulfills the inequality

$$\rho > \rho_\epsilon := \log_B[16(d+1)^2 \cdot 2^L \cdot \max(1, |x_0|)^d \cdot \epsilon^{-1}] = O(\log d + d \log \max(1, |x_0|) + L + |\log \epsilon|),$$

where we used that

$$\max(1, |m_R|)^d \leq \max(1, |x_0| + B^{-\rho}) \leq \max(1, |x_0|)^d \cdot (1 + 1/d^2)^d \leq 2 \max(1, |x_0|)^d$$

for any $\rho > 2 \log d$. Since we double ρ in each step, this shows that we succeed for a precision $\rho < 2\rho_\epsilon$. We fix this result, which will turn out to be useful at several places in the following considerations.

Theorem 1.3.10. *Let $f \in \mathbb{C}[x]$ be a polynomial of degree d with coefficients of absolute value less than 2^L, with $L \in \mathbb{Z}_{\geq 0}$, and let x_0 be an arbitrary complex value. For any non-negative integer ℓ, we can compute an approximation $\tilde{y}_0$ of $y_0 = f(x_0)$ with $|y_0 - \tilde{y}_0| < 2^{-\ell}$ using fixed point interval arithmetic with a precision ρ bounded by*

$$O(\log d + d \log \max(1, |x_0|) + L + \ell).$$

Notice that the above bound on ρ that is needed in the worst-case is also a (worst-case) bound on the input precision as, in each iteration, we need approximations of the coefficients of f as well as of x_0 to an error less than $B^{-\rho}$. We further remark that, as an alternative to the above approach, one could also use fixed point arithmetic directly to compute an approximation of $f(x_0)$, and to estimate the occurring error using Theorem 1.3.3. This yields a comparable bound on the needed precision in the worst case. However, the main drawback of this approach is that one has to work with an a priori computed worst-case error bound, which means that the needed precision is always of size $\Omega(\log d + d \log \max(1, |x_0|) + L + \ell)$. In contrast, when using interval arithmetic with increasing precision, we might already succeed with a much smaller precision.

Exercise 1.3.11. *Suppose that a polynomial $f \in \mathbb{R}[x]$ as well as a real value x_0 is given by means of an oracle that returns arbitrary good dyadic approximations of the coefficients of f and x_0. Under the assumption that $f(x_0) \neq 0$, formulate an algorithm that computes an $\ell \in \mathbb{Z}$ such that $2^{-\ell} < |f(x_0)| < 2^{\ell+2}$. How does its running time depend on $|f(x_0)|$?*

1.3.3 Floating point arithmetic

When actually implementing algorithms, the standard approach for the approximate computation with real (complex) numbers is NOT fixed point arithmetic but *floating point arithmetic*. However, a corresponding error analysis is more delicate, and thus, for the seek of simplicity, we decided to use fixed point arithmetic as our main tool for approximate computation. For a short but self-contained introduction, we refer to the appendix of [MOS11].

As already suggested by the name, we use floating point numbers to approximate an arbitrary real numbers a to a fixed *relative* error, whereas fixed point numbers are used for approximations to a fixed *absolute* error. By abuse of notation, we also use $\mathbb{F}$ to denote the set of floating point numbers with precision ρ and base B:

$$\mathbb{F} := \{-1^s \cdot m \cdot B^e : s \in \{0,1\},\ m \in \{1,\dots,B^\rho - 1\} \text{ and } e \in \{-e_{\min},\dots,e_{\max}\}\}$$

We call m the mantissa and e the exponent. Many hardware floating point units use the so-called IEEE 754 standard (1985) with *single* or *double precision*. For double precision, we have $B = 2$, $\rho = 52$, and $e \in \{-1023,\dots,0,\dots,1023\}$. There also exist several libraries for arbitrary precision arithmetic (e.g. MPFR[1]) that allow us to compute with floating point numbers of arbitrary length and (almost) no restrictions on the exponent e. In a completely analogous way as for fixed point numbers, we define the rounding functions fl(.), fld(.), and flu(.). While we have $|x - \mathrm{fl}(x)| \le B^{-(\rho+1)}$ for fixed point numbers with precision ρ (and base B), it now holds for floating point numbers of corresponding length that

$$|x - \mathrm{fl}(x)| \le 2^{-(\rho+1)} \cdot \min(|x|, |\,\mathrm{fl}(x)|).$$

Arithmetic operations of floating point numbers are defined in exactly the same way as for fixed point numbers. Furthermore, all concepts as introduced in the previous section for fixed point arithmetic also carry over to floating point arithmetic. Here, we only give the following result from [MOS11], which provides a bound on the error when using floating point arithmetic to evaluate a multivariate polynomial at a specific point. You may consider this as the counterpart of Theorem 1.3.3, which gives a comparable bound for evaluating a univariate polynomial using fixed point arithmetic.

Theorem 1.3.12. *Let $f = \sum_\alpha c_\alpha \cdot \mathbf{x}^\alpha \in \mathbb{R}[\mathbf{x}] = \mathbb{R}[x_1,\dots,x_n]$ be a polynomial of total degree d, and let $c_f := \sum_\alpha \max(1, |c_\alpha|)$ and $m_f := |\{\alpha : c_\alpha \neq 0\}|$. Then, for arbitrary real values $x_1,\dots,x_n$ of absolute value at most 2^Γ, with $\Gamma \in \mathbb{N}$, it holds that*

$$|f(x_1,\dots,x_n) - f_{\mathbb{F}}(x_1,\dots,x_n)| \le c_f \cdot (m_f + 2d) \cdot 2^{d\Gamma} \cdot 2^{-\rho},$$

where $f_{\mathbb{F}}(x_1,\dots,x_n)$ denotes the result of evaluating f at $\mathbf{x} = (x_1,\dots,x_n)$ with floating point arithmetic with precision ρ (and base $B = 2$).

The above theorem also generalizes to complex values x_i and polynomials defined over the complex numbers. The obtained error bound is comparable, that is, it only differs by a multiplicative constant from the above bound.

[1] http://www.mpfr.org/

1.4 Division

In the previous sections, we have shown how to efficiently carry out additions and multiplications on integers. We also considered corresponding operations on fixed-point numbers and intervals and estimated the error that occurs when using approximate instead of exact arithmetic. So far, any such treatment for the division of integers or fixed-/floating-point numbers a and b is missing. We will first show how to compute an arbitrary good dyadic approximation $\tilde{q} \in \mathbb{D}$ of a rational number $q := \frac{a}{b} \in \mathbb{Q}$ using only additions and multiplications of integers. We start with the special case, where $a = 1$ and b is a positive integer of length less than n. The crucial idea underlying the approach is to consider q as the unique solution of the equation $f(x) := \frac{1}{x} - b = 0$ and to use the Newton-Raphson method to derive an approximation of q. That is, with $x_0 := 2^{-\lceil \log b \rceil} \in \mathbb{D}$, we define

$$x_{i+1} := x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{\frac{1}{x_i} - b}{-\frac{1}{x_i^2}} = 2 \cdot x_i - b \cdot x_i^2 = x_i \cdot (2 - b \cdot x_i) \in \mathbb{D} \quad \text{for } i \in \mathbb{N}_{\geq 1}.$$

The first part of the following exercise shows that the sequence x_i converges *quadratically* to q. Roughly speaking, this means that the number of correct digits doubles in each iteration. We then conclude that, after $\lceil \log L \rceil$ iterations, we have computed a dyadic approximation $\tilde{q}$ of $q = 1/b$ with $|q - \tilde{q}| < 2^{-L}$. However, there is a small problem with this approach, namely, the lengths of the dyadic numbers x_i double in each iteration, and since x_0 has length $\lceil \log B \rceil \leq n$, we end up with dyadic numbers of length $O(nL)$ after $\lceil \log L \rceil$ iterations. In Part (c) of the exercise, we show that we can improve upon this approach by rounding the result obtained in the i-th iteration to the ρ_i-th digit after the binary point, with $\rho_i := 2^{i+1} + 2n$. As a result, we can reduce the length of the occurring numbers from $O(nL)$ to $O(n + L)$.

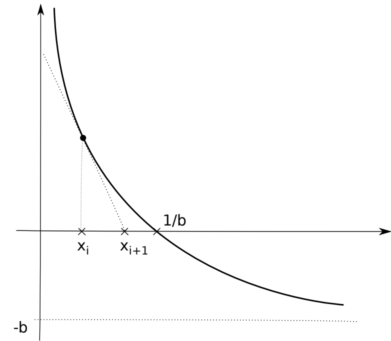

Figure 1.1: The graph of the function $f(x) = \frac{1}{x} - b$. The value x_{i+1} results from applying one step of the Newton-Raphson method to x_i.

Exercise 1.4.1. *Let $(x_i)_i$ be defined as above and L be an arbitrary positive number. Show that, for all i, it holds that*

(a) $\left|x_{i+1} - \frac{1}{b}\right| \leq b \cdot \left|x_i - \frac{1}{b}\right|^2$ *and*

(b) $\left|x_i - \frac{1}{b}\right| < \frac{1}{b} \cdot 2^{-2^i}$. *In particular, it holds that* $\left|x_i - \frac{1}{b}\right| < 2^{-L}$ *for all* $i \geq \log L$.

(c) Suppose now that we start with $y_0 := x_1 = 2^{-\lceil \log b \rceil} \cdot (2 - b \cdot 2^{-\lceil \log b \rceil})$ *and define*

$$y_{i+1} := \mathrm{fl}(y_i \cdot (2 - b \cdot y_i)) \quad \text{for } i \in \mathbb{N}_{\geq 1},$$

where we consider rounding to the nearest fixed-point number of precision $\rho_i := 2^{i+1} + 2n$. *Then, it holds* $\left|y_i - \frac{1}{b}\right| < \frac{1}{b+1} \cdot 2^{-2^i}$ *for all* i.

Hint: For (c), use that the error $2^{-\rho_i - 1}$ that is induced by the rounding in the $(i+1)$-st iteration is smaller than $\frac{2^{-2^{i+1}}}{(b+1)^2}$. Then, use induction on i to prove the claim.

Algorithm 5: Division

Input : Two non-negative n-digit integers a and b and a non-negative integer L.
Output: A dyadic number $\tilde{q} \in \mathbb{D}$ of length $O(n+L)$ such that $|\tilde{q} - a/b| < 2^{-L}$.

1 $L' := \lceil \log a \rceil + L + 1$
2 $N := \lceil \log L' \rceil$
3 $x_0 := 2^{-\lceil \log b \rceil} \cdot (2 - b \cdot 2^{-\lceil \log b \rceil})$
4 **for** $i = 1, \ldots, N-1$ **do**
5 Recursively define
6
$$x_{i+1} := \text{fl}(x_i \cdot (2 - b \cdot x_i)),$$
where fl(.) is defined as "rounding to the nearest element" in $\mathbb{F}_{2,\rho_i}$ and $\rho_i := 2^{i+1} + 2n$.
7 Compute $\tilde{q} := \text{fl}(a \cdot x_{N-1})$, where fl(.) is defined as rounding to the nearest in $\mathbb{F}_{2,L}$.
8 **return** $\tilde{q}$

From the above consideration, we conclude that we can compute a dyadic number $\tilde{q}$ with $|\tilde{q} - 1/b| < 2^{-L}$ using $O(\log L)$ additions and multiplications of integers of length $O(L+n)$. Now, computing a corresponding approximation $\tilde{q}$ of $q := \frac{a}{b}$, with integers a and b of length less than n, is straightforward; see Algorithm 5. Namely, we first compute a dyadic q' of length $O(L+n)$ such that $|q' - 1/b| < 2^{-L-\lceil a \rceil - 1}$ and then determine the product $a \cdot q'$. The result is eventually rounded to the L-th digit after the binary point. The so-obtained $\tilde{q} = \text{fl}(a \cdot q')$ has length $O(n+L)$ and it holds that $|\tilde{q} - q| < 2^{-L}$. We fix this result:

Theorem 1.4.2. *Let a and b be integers of length n. For any non-negative L, Algorithm 5 computes a dyadic approximation $\tilde{q} \in \mathbb{D}$ of length $O(n+L)$ such that $|\tilde{q} - q| < 2^{-L}$. For this, it uses $O(\log(n+L))$ additions and multiplications of $O(n+L)$-digit integers.*

We can now go one step further and derive a bound on the cost for computing an approximation of the quotient of two arbitrary complex numbers $a = a_0 + \mathbf{i} \cdot a_1$ and $b = b_0 + \mathbf{i} \cdot b_1$. Here, we assume that, for any $L' \in \mathbb{N}$, we can ask for dyadic approximations $\tilde{a}, \tilde{b} \in \mathbb{D}$ such that $|a - \tilde{a}|, |b - \tilde{b}| < 2^{-L'}$. Notice that

$$\frac{a}{b} = \frac{a_0 + \mathbf{i} \cdot a_1}{b_0 + \mathbf{i} \cdot b_1} = \frac{(a_0 + \mathbf{i} \cdot a_1) \cdot (b_0 + \mathbf{i} \cdot b_1)}{(b_0 + \mathbf{i} \cdot b_1) \cdot (b_0 - \mathbf{i} \cdot b_1)} = \frac{(a_0 b_0 - a_1 b_1) + \mathbf{i} \cdot (a_1 b_0 + a_0 b_1)}{|b|^2},$$

thus we can restrict to quotients of real numbers $a, b \in \mathbb{R}_{\neq 0}$. Suppose that dyadic approximations $\tilde{a}, \tilde{b} \in \mathbb{R}_{\neq 0}$ with $|a - \tilde{a}|, |b - \tilde{b}| < 2^{-L'} < |b|/2$ are given. Then, we have

$$\left| \frac{\tilde{a}}{\tilde{b}} - \frac{a}{b} \right| = \left| \frac{b\tilde{a} - a\tilde{b}}{b\tilde{b}} \right| = \frac{\left| b(\tilde{a} - a) - a(b - \tilde{b}) \right|}{\left| b^2 + b(b - \tilde{b}) \right|} < 2^{-L'+1} \cdot \frac{|a| + |b|}{|b|^2} \leq 2^{-L'+2} \cdot \frac{\max(|a|, |b|)}{\min(1, |b|)^2}.$$

For $L' > L + \lceil \log \max(1, |a|) \rceil + 3\lceil |\log b| \rceil + 3$, this implies that $\left| \frac{\tilde{a}}{\tilde{b}} - \frac{a}{b} \right| < 2^{-L-1}$. Hence, we may first consider L'-digit approximations $\tilde{a}, \tilde{b} \in \mathbb{D}$ of a and b, and then compute an $(L+1)$-digit approximation $\tilde{q} \in \mathbb{D}$ of their quotient $q = \frac{\tilde{a}}{\tilde{b}}$ using the method from above. Then, it holds that $|\tilde{q} - a/b| < 2^{-L}$. We fix this result:

Theorem 1.4.3. *Let $a, b \in \mathbb{C}$ be arbitrary complex numbers and $L \in \mathbb{N}$. Then, there exists a positive integer L' of size*

$$L' := O(L + \lceil \log \max(1, |a|) \rceil + \lceil |\log |b|| \rceil)$$

such that we can compute a fixed point number $\tilde{q} \in \mathbb{F} + \mathbf{i} \cdot \mathbb{F}$ of length L' with $|\tilde{q} - q| < 2^{-L}$ using $O(\log L')$ additions and multiplications of $O(L')$-digit integers. The values a and b need to be approximated to an error of size $2^{-L'}$.

Exercise 1.4.4. *For arbitrary $x \in \mathbb{R}$ with $0 \leq x \leq 1$, it holds that*

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \tag{1.7}$$

Now, for given $L \in \mathbb{N}$, use the above formula and the fact (due to Euler) that

$$\pi = 20 \cdot \arctan(1/7) + 8 \cdot \arctan(3/79)$$

to derive an efficient algorithm (i.e. with a running time polynomial in L) for computing a fixed point approximation $\tilde{\pi}$ (wrt. base 2) of π to an error less than 2^{-L}.

Hint: Estimate the error when considering only the first k summands in (1.7). Then, proceed with a suitably truncated series.

Exercise 1.4.5. *For arbitrary $x \in \mathbb{R}$ with $0 \leq x \leq 1$, we have*

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots$$

For fixed $n \in \mathbb{N}_{\geq 8}$ and arbitrary $L \in \mathbb{N}$, formulate an efficient method to compute an L-digit approximation $\tilde{\omega}$ of $\omega := \cos(2\pi/n)$.

Hint: Proceed similar as in Exercise 1.4.4 and use a sufficiently good approximation $\tilde{\pi}$ of π. For the evaluation of the truncated series at $x = \tilde{\pi}$, use Theorem 1.3.3.

Bibliography

[MOS11] Kurt Mehlhorn, Ralf Osbild, and Michael Sagraloff. "A general approach to the analysis of controlled perturbation algorithms". In: *Comput. Geom.* 44.9 (2011), pp. 507–528 (cit. on p. 14).

[MS08] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox.* SpringerLink: Springer e-Books. Springer, 2008. ISBN: 9783540779773 (cit. on p. 5).