EXACT COMPUTING IN POSITIONAL WEIGHTED SYSTEMS

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT AT CANTERBURY

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Alexander Kaganovsky

March 2001

Dedicated to my Parents

Numerical precision is the very soul of science.

— Sir D'Arcy Wentworth Thompson, *On Growth and Form* (1942)

Finite numbers can be really enormous, and the known universe is very small. Therefore the distinction between finite and infinite is not as relevant as the distinction between realistic and unrealistic.

— Donald E. Knuth [44]

At least one good reason for studying multiplication and division is that there is an infinite number of ways of performing these operations and hence there is an infinite number of PhDs (or expense-paid visits to conferences in the U.S.A.) to be won from inventing new forms of multiplier.

— Alan Clements, *The Principles of Computer Hardware*, 1986

Abstract

This thesis presents a framework which allows one to perform infinite precision numerical computations using an arithmetic based upon the representation of computable numbers by lazy infinite sequences of digits in a redundant positional radix system. We discuss advantages and problems associated with this representation, and develop well-behaved algorithms for a comprehensive range of numeric operations, including the four basic operations of arithmetic, and a number of important elementary functions. We investigate the system of real numbers represented in an arbitrary radix r, and then show that the radix-r algorithms also lend themselves with little modification to the unified representation of complex numbers in an imaginary radix, which significantly speeds up exact complex number manipulations. A full complexity analysis is given, which suggests that notwithstanding an earlier claim, positional number system representations can lead to efficient implementations of constructive arithmetic, and in particular, our algorithms largely overcome what has been known as the granularity effect. The algorithms have been implemented both in the functional programming language Miranda and imperative language C, and guidelines have been provided for optimizing and improving the existing implementations.

Contents

List of Tables

List of Figures

Trademarks

Miranda is a trademark of Research Software Ltd.

UNIX is a registered trademark of The Open Group in the United States and other countries.

FreeBSD is a registered trademark of Walnut Creek CDROM.

Sun, Sun Workstation, Java, Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

All SPARC trademarks are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

AMD-K6-2 is a registered trademark of Advanced Micro Devices, Inc. in the United States.

SGI is a trademark of Silicon Graphics, Inc.

StarCraft is a trademark of Davidson & Associates, Inc. in the United States and other countries.

All other trademarks and registered trademarks are the property of their respective holders.

Notational conventions

$\mathbb{N}$	the set of natural numbers $\{1, 2, 3, \cdots\}$
$\mathbb{N}_0$	the set of natural numbers and zero $\{0, 1, 2, 3, \cdots\}$
$\mathbb{Z}$	the ring of integers $\{0, \pm 1, \pm 2, \pm 3, \cdots\}$
$\mathbb{Q}$	the field of rational numbers
$\mathbb{R}$	the field of real numbers
$\mathbb{C}$	the field of complex numbers
$\lfloor x \rfloor$	greatest integer not greater than x
$\lceil x \rceil$	smallest integer not less than x
$\{x\}$	the fractional part of x
A^B	the set of all functions from B to A

Acknowledgements

The author gratefully acknowledges the financial support of the Computing Laboratory at the University of Kent at Canterbury, and the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom. Partial support for this research was also provided by grants from the Ian Karten Charitable Trust and the Anglo-Jewish Association. During the writing-up period, the author was supported by EPSRC, Grant Reference GR/L03279.

I would like to express my heartfelt thanks to David Turner for always offering help whenever I needed it, for the many interesting and informative discussions, and numerous suggestions and guidance in this project all the way from inception to completion. No-one could possibly have a better supervisor.

Credit for some of the ideas put forward in this thesis must go to Carl Pixley, whose early unpublished work at Burroughs Corporation's Austin Research Centre has strongly influenced my research and was a great source of inspiration. I owe much to members of the COMPROX group for a great deal of valuable and pertinent comment, and also to the many people whose ideas have been consciously and unconsciously incorporated. I also wish to thank an anonymous ENTCS referee for his careful reading of my paper [39] and pointing out a number of errors and obscurities. I was fortunate to be able to make extensive use of the Templeman Library at the University of Kent, and the enormous resources of the British Library, for which I am very grateful.

I gladly record my obligation to Alister, Dennis, Dimmy, Kevin and the many friends who have borne charitably with me during the somewhat excessive period of gestation of this thesis, and who provided numerous interesting diversions from the tedium of writing. A special acknowledgement goes to Miguel for his help in the final stages of the preparation and submission of this thesis.

I am very grateful to my parents for their incessant support, understanding and not asking. Their love and encouragement helped make this thesis possible.

Last but not least, I give my Thanks and Praise to Jah, for his guidance and protection, for creating the Universe, the sunshine, the rain, and all things good, bad, and crazy. Without Him, this work would not have been possible (as nothing else would).

Chapter 1

Introduction

Press $\boxed{1/x}$ to invert the matrix.
Note that matrix inversion can produce erroneous results
if you are using ill-conditioned matrices.
— HEWLETT-PACKARD, *HP* 48G Series User's Guide (1993)

1.1 What is exact computing?

The standard implementations of real numbers on a computer are approximations held to some fixed number of significant figures. The accumulation of round-off errors leads to well-known difficulties calculating accurate numerical results for scientific and engineering problems. Going to double, quadruple or even multiple precision in no way eliminates these problems, but merely ameliorates them. No matter how much precision is provided, there are always problems for which it is insufficient to produce reliable results. Perhaps one of the worst features of floating point arithmetic is that the computer can give us no indication of how many of the digits printed are actually meaningful, so with a poor choice of algorithm it is quite easy to generate numerical answers that are completely meaningless.

As computing power becomes cheaper, it seems reasonable that we may wish to move to a form of real arithmetic that is perhaps more expensive but which will generate results to numerical calculations that carry with them some easily understood guarantee of accuracy. Modern functional programming languages provide certain computing abstractions — infinite lists, higher order functions — which make it possible to represent real numbers exactly as they are defined in mathematics, using any of several possible methods.

Mathematically a real number is defined as an infinitary object — for example, a converging sequence of rationals — but there are many other different representations. Since all our computers are finite, it stands to reason that only finitely many entries of an infinite sequence can be instantiated in finite time. It also follows that not all real numbers can be represented on a computer — only those whose defining sequence can be determined by a *finite* amount of information.

Is it possible to define mathematical operations on the set of number representations so that the results of such operations would faithfully represent the results of the matching mathematical operations on real numbers? If such were possible, a number representation would have a precise mathematical meaning, and we would be able to operate on representations just as we do on numbers, with fundamental mathematical laws preserved. What subset of the real numbers can be thus represented? What predicates are definable on number representations that represent the correspondent predicates on the reals?

The purpose of this thesis is to address all these questions, and investigate the properties of the so-called redundant signed-digit positional representation of real and complex numbers in order to find whether it can be rendered free from the objections which have caused its rejection by the majority of the researchers, who have deserted altogether its line of approach (see Section 1.6). In so doing, we develop algorithms for a wide range of numerical operations which largely overcome what has been known as the granularity effect, discuss complexity issues, and examine various factors that can affect implementations.

1.2 How this thesis is organized

This thesis is divided into five chapters. Chapter 1 is intended to be an introduction to the subjects discussed, and provides an overview of number systems, representations, computability and previous work on the subject. Chapter 2 is devoted to exact real arithmetic in a radix-r system, initially focusing on the normalization function, and then detailing the four operations of arithmetic, square root and a number of transdendental functions. It is also the most comprehensive chapter, which lays the foundation of knowledge for those following. Chapter 3 adapts the algorithms to also work with exact complex numbers using the imaginary radix representation of same. Chapter 4 includes a complexity analysis of the algorithms, as well as a discussion of implementation details in functional and imperative languages. Chapter 5 reviews the work and draws conclusions.

There are three appendices. The first is the Miranda source code for the exact real arithmetic package. The second appendix is the Miranda source code for complex arithmetic. The last appendix is the C source code rewrite of some of the Miranda functions.

1.3 The need for exact real arithmetic

To demonstrate the inadequacy of floating-point arithmetic, let us consider the following innocuous-looking calculation:

```
for (i=1; i<=62; i++)
  x = sqrt(x);
for (i=1; i<=62; i++)
  x = x*x;
```

This computation involves no subtractive cancellation, and all the intermediate numbers lie between 1 and x, so one might expect it to return an accurate floating-point approximation to x. However, instead of $f(x) = x$, the algorithm computes, in double precision floating point, the function

$$f'(x) = \begin{cases} 0, & 0 \le x < 1 \\ 1, & x \ge 1. \end{cases}$$

How can this happen in just 124 operations on non-negative numbers?

This calculation simply exhausts the accuracy and range of a machine with double precision — if the positive numbers x representable on the computer at double precision satisfy $4.94 \cdot 10^{-324} \le x \le 1.79 \cdot 10^{308}$ (IEEE-754), and $r(x) = x^{1/2^{62}}$, then for any machine number $x \ge 1$, we have

$$\begin{aligned} 1 \le r(x) < r(10^{308}) &= 10^{308/2^{62}} = e^{308 \cdot 2^{-62} \cdot \ln 10} \\ < e^{10^{-15}} &= 1 + 10^{-15} + \frac{1}{2} \cdot 10^{-30} + \ldots, \end{aligned}$$

which the repeated squarings quickly turn into 1.

One common misconception about floating-point arithmetic is that poor accuracy is usually caused by the accumulation of millions (or a very large number) of rounding errors. Another misconception is that any desired accuracy is achievable by computing at a high enough precision, and comparing how many digits of the original and the (presumably) more accurate answer agree[1].

The following example, due to J.-M. Muller [53], should dispel both of these misconceptions, providing an unsettling glimpse of how an ostensibly straightforward floating-point computation of a sequence of numbers can run foul of rounding anomalies and produce completely erroneous results that differ from the exact ones by several orders of magnitude after only 14 divisions and 12 subtractions.

Let $(a_n)_{n \in \mathbb{N}_0}$ be the sequence defined recursively as follows

$$a_0 = \frac{11}{2}, \quad a_1 = \frac{61}{11}, \quad a_{n+1} = 111 - \frac{1130 - 3000/a_{n-1}}{a_n}. \tag{1}$$

[1] Although this may be true of most algorithms, there is no guarantee that a result computed in m digit precision will be any more accurate than one computed in n digit precision, for $m > n$; for instance, both results could have no correct digits.

In exact arithmetic the a_n form a monotonically increasing sequence that converges to 6, which should become obvious if we notice that

$$a_n = \frac{6^{n+1} + 5^{n+1}}{6^n + 5^n}.$$

However, if we try to implement recurrence (1) to compute the successive terms of this sequence in single-precision floating point arithmetic, we obtain the following values

n	6	7	8	9	10	11	12	13
a_n	5.68	4.58	-20.51	134.13	101.49	100.09	100.01	100.00

(2)

which suggest that the sequence converges to 100.

How can so simple a sequence produce such an inaccurate result after only 26 elementary operations? The reason the above sequence malfunctions is simply a result of the fact that the function

$$f(x) = 111 - \frac{1130 - 3000/x}{x}$$

has the numbers 5, 6 and 100 as its fix points, of which 100 is an attractor. Rounding errors in the representation of a_1 on a binary machine cause the computed values of a_n to be of the form

$$a'_n = \frac{100^{n+1}\varepsilon + 6^{n+1} + 5^{n+1}}{100^n\varepsilon + 6^n + 5^n},$$

for a constant ε of order the unit roundoff.

Can this pathology be remedied by going to multiple precision? The answer in this case is no — it would merely delay it. Even if rounding is only slightly sloppy, the sequence will inevitably converge to 100 (and rather rapidly at that — with k decimal digits of precision, the value of a_{2k} is almost guaranteed to be close to 100). This clearly demonstrates that for a very ill-conditioned problem, instability can be caused by the insidious growth of just *a few*, and not necessarily a large number of rounding errors.

The foregoing examples serve to illustrate the inherent inaccuracy and hazards that beset the use of floating-point (including multiple-precision) numbers in numerical computations. An awareness of error and the methods of dealing with it are essential to the confidence in computation techniques for both numerical analysts and end users alike, as error in floating-point calculations can jeopardize the validity of results and undermine one's faith in computer's ability to perform computations correctly.

1.4 Numbers and number systems

1.4.1 Definitions

We shall now examine the familiar system of numeration as well as a few other number systems of interest, and also introduce some terminology used for describing these systems.

The most basic unit in a number system is the *digit* which we shall regard as having the commonly accepted interpretation, i.e. as a symbol representing a single integral quantity. The collection of digits used in a number system will hereinafter be referred to as its *digit set* or *alphabet*. Any number can then be written as a string of symbols chosen from an alphabet.

Definition 1. *An exact real number system is a pair* (A, φ)*, where* $A \subset \mathbb{Z}$ *is a countable digit set (or alphabet), and* φ *is a representation function* $\varphi : R \to \mathbb{R}$*, defined on the representation set* $R \subset A^{\mathbb{N}}$ *so that the image* $\varphi(R)$ *is dense in a neighbourhood of zero in* $\mathbb{R}$*. A representation of the number* $r \in \varphi(R) \subset \mathbb{R}$ *in the system* (A, φ) *is any member of the set* $\varphi^{-1}(r)$*.*

This definition is not specific to the real numbers — any other field[2] $\mathbb{F}$ can be

[2]Recall that a set $\mathbb{F}$ with two operations $+$ and $\cdot$ is a *field*, if $\mathbb{F}$ is a commutative group under $+$, $\mathbb{F}\backslash\{0\}$ is a commutative group under $\cdot$, and the distributive law holds.

used instead of $\mathbb{R}$, for example, the field of complex numbers $\mathbb{C}$.

Clearly, φ defines an equivalence relation in R if we specify that $a \sim_\varphi b$ if and only if $\varphi(a) = \varphi(b)$. The quotient set of R relative to the relation $\sim_\varphi$ (the set of equivalence classes of inverse images of elements in $\varphi(R)$) will be denoted by $\bar{R}$:

$$\bar{R} = R/\sim_\varphi .$$

There is a natural map $\bar{\varphi}$ of $\bar{R}$ into $\mathbb{R}$: abbreviating the equivalence class containing $a \in R$ to $\bar{a}$, we simply define $\bar{\varphi}$ by

$$\bar{\varphi}(\bar{a}) = \varphi(a).$$

Since $\bar{a} = \bar{b}$ if and only if $\varphi(a) = \varphi(b)$, it is clear that the right-hand side is independent of the choice of the element a in $\bar{a}$ and so, indeed, we do have a map. It is also clear that $\bar{\varphi}$ is injective (distinct elements of $\bar{R}$ have distinct images in $\mathbb{R}$); hence, $\bar{\varphi} : \bar{R} \to \operatorname{im} \bar{\varphi}$ is a bijection (one-to-one correspondence).

So far, we have not said anything about the properties of the representation set R. Clearly, we would like R to inherit as many properties of $\mathbb{R}$ (or field $\mathbb{F}$) as possible, and in particular, we would like to

1. Define binary operations $\oplus$, $\ominus$, $\otimes$, and $\oslash$ on R that are preserved by φ so that R becomes a field:

$$\begin{aligned}
\varphi(a \oplus b) &= \varphi(a) + \varphi(b)\\
\varphi(a \ominus b) &= \varphi(a) - \varphi(b)\\
\varphi(a \otimes b) &= \varphi(a) \cdot \varphi(b)\\
\varphi(a \oslash b) &= \varphi(a)/\varphi(b)
\end{aligned}$$

   (and, in particular, $a \oslash b$ should be defined for all $a \in R$, $b \in R \backslash \{\varphi^{-1}(0)\}$),

2. Extend the domain of function φ to include finite subsequences of R so that

for any $a = (a_n)_{n\in\mathbb{N}} \in R$,

$$\lim_{n\to\infty} \varphi(a_1, \ldots, a_n) = \varphi(a).$$

The last condition is needed to enable us to make useful inferences about a number from a finite amount of information about its representation.

Since $\bar{\varphi}$ is injective, the binary operations are easy to define on the set $\bar{R}$ of equivalence classes:

$$\bar{a} \oplus \bar{b} = \bar{\varphi}^{-1}(\varphi(a) + \varphi(b))$$

(and similarly for $\ominus$, $\otimes$, and $\oslash$). If there are similar φ-preserving binary operations on R , then for any $a \in \bar{a}$, $b \in \bar{b}$, we should have

$$a \oplus b \in \bar{a} \oplus \bar{b}$$

(similarly for $\ominus$, $\otimes$, and $\oslash$).

Positional systems

Conventional number systems are *fixed-radix positional weighted* systems for which the weight w_i of the i-th digit, a_i, is r^i, the range of each digit is $\{0, 1, \cdots, r-1\}$, and the numerical value of the string $a_{n-1}a_{n-2}a_{n-3}\cdots a_0$ is interpreted as

$$a = \sum_{i=0}^{n-1} a_i w_i = \sum_{i=0}^{n-1} a_i r^i \tag{3}$$

Our familiar number system uses $r = 10$ as the base (or *radix*, as it is also called), but any other base (except 1 or 0) can also be used. It is not necessary to restrict r to positive or whole numbers, but as a practical matter, we will only discuss radices that are positive integers.

The exponents of r in (3), of course, can be negative as well as positive, thus making the weighting factors fractional and enabling our number system to represent fractions as well as integers. It is our custom to write mixed numbers with a period intervening at the boundary between the integer and the fraction portions. In the case of a radix-10 system, the period is usually called the *decimal point*, while the general name is *radix point*.

Speaking in terms of Definition 1, a positional radix-r system is a pair (A, φ), where $A = \{0, 1, \cdots, r-1\}$, $\varphi((a_n)_{n\in\mathbb{N}}) = \sum_{i=1}^{\infty} a_i r^{-i+1}$, and $\varphi(a_1, \ldots, a_n) = \sum_{i=1}^{n} a_i r^{-i+1}$. Thus specified, the φ function is defined everywhere in $A^{\mathbb{N}}$, hence the representation set R coincides with the whole of $A^{\mathbb{N}}$.

It is somewhat surprising to note that for any radix r there exists an infinite number of digit sets. For instance, we can choose the following unusual set for radix $r = 10$:

$$A = \{\underline{0}, \underline{1}, \underline{20}, \underline{21}, \underline{40}, \underline{41}, \underline{60}, \underline{61}, \underline{80}, \underline{81}\}$$

In this set, we would represent 5 as $\underline{1}.\underline{40}$.

Continued fractions

The continued fraction system is an interesting example of a non-positional representation. Continued fractions are numbers of the form

$$a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \ddots}} = a_1 + b_1/(a_2 + b_2/(a_3 + \ldots)),$$

where a_i and b_i are integers for all $i \in \mathbb{N}$. If $b_i = 1$ for all $i \in \mathbb{N}$, the continued fraction is usually written as $[a_1, a_2, a_3, \ldots]$.

Here the digit set A is the set of all integers $\mathbb{Z}$, and the φ function is defined as above:

$$\varphi([a_1, a_2, a_3, \ldots]) = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \ddots}}. \qquad (4)$$

R is the set of all sequences $[a_1, a_2, a_3, \ldots]$ for which continued fraction (4) is convergent.

The algorithms for conversion between positional, rational, and continued fraction representations are quite simple. For instance, let $r \in \mathbb{R}$ be a number. Compute

$$r = a_0 + \frac{1}{r_1},\ r_1 = a_1 + \frac{1}{r_2},\ \ldots,\ r_k = a_k + \frac{1}{r_{k+1}}$$

where $a_k \in \mathbb{Z}$ is "close" to $r_k \in \mathbb{R}$, e.g. if we can take $a_k = \lfloor r_k \rfloor$ [87].

We can also write

$$\begin{pmatrix} r \\ 1 \end{pmatrix} = \begin{pmatrix} a_0 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} a_k & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} r_{k+1} \\ 1 \end{pmatrix}$$

or

$$r = [a_0 a_1 \cdots a_k]\, r_{k+1} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_k + \cfrac{1}{r_{k+1}}}}}$$

Continued fractions represent some numbers more easily than the positional notation. In particular,

1. Rational numbers, represented as continued fractions, always terminate (in positional notation it is base-dependent).

2. Quadratic surds (numbers of the form $(a + b/\sqrt{D})/c$, where $a, b, c, D \in \mathbb{Z}$, $b \neq 0$, $c \neq 0$, $D > 1$, and D is not a perfect square) always repeat when represented as continued fractions (in positional notation, quadratic surds are generally unrecognizable as such).

3. The usual transcendental functions of rational arguments have simple continued fraction expansions.

4. The continued fraction expansion (with positive denominators) of a real number gives the complete set of best rational approximations to the value

they represent — if we truncate a continued fraction at any point, the resulting rational number gives an approximation better than any other rational number with the same denominator. Moreover, such rational approximation will automatically be in lowest terms.

Each representation favours certain mathematical operations — for example, decimal favours multiplication by powers of 10. Continued fraction representation has its unique advantage in that reciprocation is performed with trivial effort:

$$1/[a_1, a_2, a_3, \ldots] = [0, a_1, a_2, a_3, \ldots] \tag{5}$$

Negation is also very simple and is achieved by negating all the terms, optionally observing that

$$[-a_1, -a_2, -a_3, -a_4, \ldots] = [-a_1 - 1, 1, a_2 - 1, a_3, a_4, \ldots] \tag{6}$$

Some non-conventional number systems

In this section we shall give a very brief overview of some unconventional number representations and arithmetic. In particular, we shall discuss mixed-radix number systems and negative-radix systems. A multitude of other unusual (and sometimes unuseful) number systems exist (e.g., the residue system, the logarithmic system, the binomial system, the Fibonacci system); however, we shall not have occasion to use them in the work that follows, and therefore refer the reader to the literature for more details [74].

Mixed-radix systems are positional weighted systems where each digit position i has an associated radix r_i and each digit a_i is chosen from among a set of digits allowed in position i, i.e.,

$$a_i \in \{0, 1, \ldots, (r_i - 1)\}\,.$$

The value of the number a represented by $a_n a_{n-1} \cdots a_2 a_1 a_0$ is taken as

$$a = a_0 + a_1 \cdot r_1 + a_2 \cdot r_1 r_2 + \ldots + a_n \cdot r_1 r_2 \cdots r_n$$

(r_0 is usually chosen to be zero). The usual radix-r system is a mixed-radix system with $r_1 = r_2 = \ldots = r$.

Fractions can also be represented in mixed-radix systems by extending the digit group to the right of the "radix point" and assigning radices to the negative-subscripted positions:

$$a_n a_{n-1} \cdots a_2 a_1 a_0 . a_{-1} a_{-2} \cdots a_{-n} \cdots = \sum_{k=-n}^{\infty} \left(a_{-k} \cdot \left(\prod_{i=0}^{|k|} r_i \right)^{-\operatorname{sgn} k} \right).$$

Here we let the weights of the fraction positions be

$$\frac{1}{r_1}, \; \frac{1}{r_1 r_2}, \; \frac{1}{r_1 r_2 r_3}, \; \ldots$$

Mixed-radix systems are often used in everyday life in measuring length, time, money and other quantities. For example, we measure time in seconds, minutes, hours, days and weeks, which is a mixed-radix system with the radices $r_0 = 1$, $r_1 = 60$, $r_2 = 60$, $r_3 = 24$, and $r_4 = 7$.

Another interesting example of the mixed-radix system is the *factorial number system*, in which the radix associated with digit position i is i, and therefore the weight of the i-th digit is $i!$ We shall use this system in Chapter 2 to compute the constant e, which is represented in the factorial system by the sequence

$$1.1111111\ldots = 1 + 1 \cdot \frac{1}{1!} + 1 \cdot \frac{1}{2!} + 1 \cdot \frac{1}{3!} + \ldots$$

(also see Appendix C).

Negative-radix systems are positional weighted systems for which the weight of the i-th digit is $(-r)^i$, the range of each digit is $\{0, 1, \cdots, r-1\}$, and the value of $a_{n-1}a_{n-2}a_{n-3}\cdots a_0$ is

$$a = \sum_{i=0}^{n} a_i(-r)^i.$$

The value of r most often used is 2, and the associated *negabinary* system has some advantages over the usual binary system, because it does not require a separate sign bit; instead, the sign is implied by the number itself. Numbers in radix $(-r)$ can be easily expressed in terms of the radix $(+r)$ as follows

$$\sum_{i=0}^{n} a_i(-r)^i = \sum_{i=0}^{\lfloor n/2 \rfloor} a_{2i}r^{2i} - \sum_{i=0}^{\lceil n/2 \rceil -1} a_{2i+1}r^{2i+1}.$$

Thus, all even-numbered digit positions make a positive contribution to the number, and all odd-numbered positions make a negative one.

We shall make use of negative-radix systems in Chapter 3, where we discuss complex number arithmetic.

1.5 Computability

1.5.1 Review of the "classical" real number systems

The reader will certainly be familiar with the classical construction of the *real number system*. We shall however review some of the basic definitions.

We assume as given the *natural numbers* $1, 2, 3, \ldots$ and the *integers* $0, \pm 1, \pm 2, \ldots$, and denote them by $\mathbb{N}$ and $\mathbb{Z}$ respectively. The set $\mathbb{N} \cup \{0\}$ will be denoted by $\mathbb{N}_0$. The *rational numbers* are then defined as quotients of integers or, more precisely, as equivalence classes of ordered pairs (m, n), $n \neq 0$:

$$((m_1, n_1) \sim (m_2, n_2)) \iff (m_1 n_2 = m_2 n_1)$$

Thus defined set of rational numbers we shall denote by $\mathbb{Q}$.

Given the rationals, there are two chief ways of defining *real numbers*. First method is due to Dedekind [21, 20] and defines a real number by a "cut" of all rationals in two classes such that each member of one class is less than every member of the other. In the second method (Cantor, Cauchy), real numbers are interpreted as suitably defined equivalence classes of monotone converging infinite sequences of rationals [17, 18]. It can be shown that the two methods are equivalent to each other. For example, one might define the "real decimal numbers" in terms of infinite sequences of fractions with denominators that grow by powers of 10, while the numerators grow one digit at a time. This would obviously define the same structure of the real numbers.

The natural, rational and real numbers can also be constructed axiomatically [80, 54], but this construction does not really belong to the subject of our discussion. For our purposes it will be enough to assume the existence of the set of real numbers and take the properties of this set for granted. The set of all real numbers will be denoted by $\mathbb{R}$.

1.5.2 The Turing-computable real numbers

In defining the *computable real numbers*, we shall follow Turing's original definition [83, 84], in a manner parallel to that of defining the (Cantor) real numbers as sequences of digits interpreted as decimal fractions.

Definition 2 (Turing). *A real number $x \in \mathbb{R}$ is called computable if its decimal expansion $x = a_0.a_1a_2 \ldots a_n \ldots$ can be written down by a Turing machine. That is, there exists a Turing machine that starts with a blank tape and prints out a tape of the form*

$$a_0 X a_1 X a_2 X \cdots a_n X \cdots$$

We require that once a digit has been computed and printed (which is symbolized by the printing of an X), the machine must never move to the left of, or change, that digit.

Roughly speaking, a computable real is one that can be effectively approximated to any degree of precision by a finite computer program given in advance.

When more precision is desired, the computation may take longer, but the program itself does not change. This, in fact, imposes a serious limitation on the numbers, because the defining sequence of a computable real must be determined by a *finite* amount of information, namely, the state diagram of the associated Turing machine.

It is also clear that the representation of the real numbers as sequences of *decimal* digits is not essential to the definition; we can just as readily use any other radix, as well as any other representation involving generation of an infinite sequence of numbers, e.g. continued fractions.

Worthy of mention are some other ways of looking at the definition of the computable reals. In his classical paper [19], Alonzo Church introduced the notion of "effective calculability" by identifying effectively calculable functions of positive integers with *recursive* functions of positive integers (see also [42]). His assumption — now known as "Church's Thesis" — that "effectively calculable" should be identified with the general recursive functions, turned out to be identical with Turing's assertion that any process, which could naturally be regarded as possible to carry out ("an effective procedure"), can be realized by some Turing machine having a suitable set of instructions[3] [83, 81, 22]. One cannot expect to prove or disprove Turing or Church's theses, since the term "naturally" relates to human dispositions rather than to the computation itself. Nevertheless, the years have borne out their claim and now it is generally accepted as an empirical truth. If we accept Church's thesis, we can interpret computability of a given number or other object as meaning the existence of certain recursive functions. The appropriate definition is given below.

Definition 3. *[64] A real number x is called computable if there is a computable sequence*[4] *of rationals $(r_n)_{n\in\mathbb{N}} \in \mathbb{Q}$, which converges effectively to x; this means that there is a recursive function $e(n)$ for which $k \geq e(n)$ implies $|x - r_k| \leq$*

[3]The latter proposal has become known as *Turing's thesis*

[4]*A sequence $(r_n)_{n\in\mathbb{N}} \in \mathbb{Q}$ of rational numbers is called computable if there exist three recursive functions a, b, s from $\mathbb{N}$ to $\mathbb{N}$ such that $r_n = (-1)^{s(n)}\,[a(n)/b(n)]$ for all $n \in \mathbb{N}$*

r^{-n}. Then, passing effectively to the subsequence $(r'_n)_{n\in\mathbb{N}} = (r_{e(n)})_{n\in\mathbb{N}}$, we have $|x - r'_n| \leq r^{-n}$ for all $n \in \mathbb{N}$. Here r is a radix value, e.g. $r = 10$.

For other definitions of recursive reals, see [67, 79, 69].

1.5.3 The existence of non-computable real numbers

While rational numbers are computable, it is clear that not *all* real numbers are; at least, because the set of all computer programs (or the set of the tapes for a universal Turing machine) is countable, whereas it is a well-known fact (see e.g. [17]) that the set of all real numbers is not. Hence there must be some non-computable real numbers, and indeed it follows that *almost all* real numbers turn out to be non-computable.

However, the class of computable numbers, although enumerable, is very dense and in many ways similar to the classical continuum. It includes the numbers π, e, the real parts of all algebraic numbers, the real parts of the zeros of the Bessel functions and, basically, all well-known numbers. It doesn't however include all *classically definable* numbers: it is possible to classically define a number which is not computable, although a constructive mathematician would not recognize such a definition as succeeding in defining a real number.

At this point, it would be well to distinguish the concepts of *constructive* analysis and *recursive* analysis.

Constructive analysis admits no real numbers other than computable ones and no methods of proof other than constructive ones [1, 9, 16, 36, 71, 92]. The notion of "computable function" or "rule" is not defined within the framework of the theory and is considered to be a primitive one. In particular, computability is not necessarily to be identified with Turing-computability[5].

On the other hand, the subject of study of *recursive analysis* [67, 64, 65, 10, 33, 32], is a certain classically defined subset of the real numbers, called the *recursive*

[5] It is interesting to note that Cantor's proof that the real numbers are uncountable remains valid in constructive analysis.

reals. In recursive analysis, "computable" is simply a synonym for "recursive" and is a defined idea. In recursion theory, it is usual practice to make free use of "non-constructive" methods to delineate the set of computable numbers within the larger set of all real numbers (or the set of computable functions within that of all functions).

For better visualization of these ideas, we shall adopt the viewpoint of recursion theory, and give an example of a particular non-computable real number R_U.

In our construction, we shall appeal to the unsolvability of the halting problem for Turing machines (it was first established by Turing [83], see also [22]). That is, given a Turing machine $\mathbf{M}$ and an initial tape T, no Turing machine can compute the function

$$h(\mathbf{M}, T) = \begin{cases} 1, & \text{if Turing machine } \mathbf{M} \text{ halts for initial tape } T \\ 0, & \text{otherwise} \end{cases}$$

Now let us consider the behaviour of a Turing machine $\mathbf{M}$ on some tape T inscribed with a finite sequence composed of the ten symbols $0, 1, \ldots, 9$. The number of such tapes is countably infinite, because we can set them into one-to-one correspondence with natural numbers. We now define the number R_U in terms of the enumerated set of tapes:

$$R_U = x_0.x_1x_2 \ldots x_n \ldots$$
$$x_n = \begin{cases} 1, & \text{if Turing machine } \mathbf{M} \text{ halts on the } n\text{-the tape} \\ 0, & \text{otherwise} \end{cases}$$

The real number thus defined is obviously non-computable, because if there existed a Turing machine which could compute it, we could have built from it a somewhat more complicated machine which can decide whether or not $\mathbf{M}$ halts on an arbitrary tape T. However, as we have pointed out, no such machine exists.

From a constructive point of view, this is not a valid definition of a real number, because no directions have been given for calculating it.

Another interesting property of the computable reals is that, while being enumerable, they cannot be "effectively enumerated", i.e. arranged in a sequence which can be represented by a Turing machine [83].

1.5.4 Computability and representations of numbers

In classical mathematical analysis real numbers are defined in a variety of ways, such as converging sequences of rationals, infinite strings of decimal (or radix-r) digits, infinite fractions, etc. From the classical point of view, all these ways are equivalent to each other, and the choice of a particular representation is matter of convenience. It may therefore come as a surprise to some to learn that some functions on the computable reals, such as the four basic arithmetic operations, are critically dependent on the representation, and with a poor choice of the latter may become non-computable.

The following example is borrowed from [55] (with a slight modification). We shall use the representation of real numbers by *conventional* decimal expansions, i.e. infinite sequences of digits in the range $[0, 9]$, and show that the addition operation is non-computable in certain cases. Specifically, we shall present two numbers a and b having respective expansions $(a_n)_{n\in\mathbb{N}_0}$ and $(b_n)_{n\in\mathbb{N}_0}$ such that even the first digit of the conventional expansion of $a + b$ cannot be computed.

In so doing, we shall use our ignorance of the behaviour of the decimal expansion of π. Namely, nobody knows whether a sequence 12345678 occurs in that expansion[6]. We shall therefore call k the *critical number* of π, if 12345678 occurs in its decimal expansion for the first time beginning at the k-th place. Certainly, for any number $n \in \mathbb{N}$ we can determine whether or not n is critical: all we have to do is to compute π to the first $n + 7$ decimal places.

[6]A search of the first 10 million digits of Pi did not produce any matches (see `http://gryphon.ccs.brandeis.edu/~grath/attractions/gpi/`), however π is believed to be uniform in all bases, so in this particular example we do know that 12345678 occurs in π. The example, however, only serves to illustrate the fact that the online sum could not be computed if the existence of k were proved undecidable.

Let us now define

$$a_0 = 0,$$

$$a_n = \begin{cases} 3, & \text{if } n \text{ is not the critical number of } \pi \\ 4, & \text{if } n = 2k+1,\ k \in N \text{ is the critical number of } \pi \end{cases},\ n \in \mathbb{N}$$

$$b_0 = 0,$$

$$b_n = \begin{cases} 6, & \text{if } n \text{ is not the critical number of } \pi \\ 5, & \text{if } n = 2k,\ k \in N \text{ is the critical number of } \pi \end{cases},\ n \in \mathbb{N}$$

Thus, if the critical number k of π is odd, $a > \frac{1}{3}$, $b = \frac{2}{3}$, and $a + b > 1$; if k is even, we have $a = \frac{1}{3}$, $b < \frac{2}{3}$, $a + b < 1$; and if k does not exist, i.e. no 12345678 occurs in the decimal expansion of π, then $a = \frac{1}{3}$, $b = \frac{2}{3}$, $a + b = 1$.

If we could compute even the first decimal place of $a + b$ (0 or 1), we could prove one of the two propositions: "if k exists, it is odd" or "if k exists, it is even". Since we have not proved either of the two, we cannot write down even one place of $a + b$. Of course, it may well happen that k does exist and so this particular problem can be solved by the "brute force" method, i.e. computation of the digits of π to enormous precision. But if such k does not exist, the search will be to no avail and the computation will never terminate.

1.5.5 Online algorithms and redundant number systems

Although the example of the previous section was somewhat discouraging, it should at any rate have served to warn the reader that not all number systems are suitable for infinite precision computations. We proceed now to ease the situation by proving a theorem about some basic properties a number system must possess to enable us to perform certain natural operations on number representations. Namely, we would like to be able to generate the first digit of a result by only looking at a finite number of the operand digits, and produce more digits of the result by looking at successive operand digits with a bounded delay.

Definition 4. *An algorithm that works on number representations (sequences in $A^{\mathbb{N}}$) is said to be an online algorithm if to generate digit k of the result it is sufficient to have available the leading $k + \delta$ digits of the operands, where δ is a positive integer. The constant δ is known as the online delay and should be reasonably small for a good algorithm (for common operations, δ is typically between one and four).*

More information about online arithmetic in finite systems can be found e.g. in [27].

A number system is said to be *redundant* if there are at least two distinct sequences that are mapped onto the same number; otherwise, it is *non-redundant*. For example, a radix-r number system requires *at least* r digit symbols; if this number is greater than r, the system becomes redundant. Notice that even the conventional decimal system exhibits some form of redundancy — for example,

$$1/2 = 0.50000\ldots = 0.49999\ldots,$$

although it is not sufficient to effectuate online computation.

We shall now proceed to prove that if one can add infinite representations of exact real numbers (see Definition 1) in an online manner, the number system must be *fully redundant*, as defined below.

Firstly, we shall establish a connection between the redundancy of a number system and its representation set, and give an equivalent definition of a fully redundant system in terms of representations sets.

For any number representation $a = (a_1, a_2, \ldots)$ we can consider the following sets:

$$\Phi^{(n)}(a_1, \ldots, a_n) = \{\varphi(x) \mid x = (a_1, \ldots, a_n, x_{n+1}, x_{n+2}, \ldots) \in R\},$$

where $x_k \in A$ are arbitrary numbers (for each $n \in \mathbb{N}$, the set $\Phi^{(n)}(a_1, \ldots, a_n)$ contains the numbers represented by sequences that begin with the n digits

$(a_1, a_2, \ldots, a_n)$.

Definition 5. *A number system (A, φ) is fully redundant if there is a $k \in \mathbb{N}$ such that for any $a = (a_1, \ldots, a_n, \ldots) \in R$ and $n \geq k$ there exists a sequence $b = (b_1, \ldots, b_n)$ different from $(a_1, \ldots, a_n)$ ($\exists i$ with $1 \leq i \leq n$ and $b_i \neq a_i$) with*

$$\Phi^{(n)}(a_1, \ldots, a_n) \cap \Phi^{(n)}(b_1, \ldots, b_n) \neq \emptyset.$$

This means that for any $n \in \mathbb{N}$ there are numbers (belonging to the above intersection) that have representations with different prefixes $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$. Note that this requirement is weaker that the requirement that at least two distinct sequences be mapped onto each number (in a fully redundant system some numbers may have a unique representation).

In what follows, we shall deal with number systems that satisfy two additional properties:

1. For all $n \in \mathbb{N}$ and $(a_k)_{k \in \mathbb{N}}$ the set $\Phi^{(n)}(a_1, \ldots, a_n)$ is bounded

2. For all $(a_k)_{k \in \mathbb{N}}$,

$$\Phi^{(n)}(a_1, \ldots, a_n) \supset \Phi^{(n+1)}(a_1, \ldots, a_n, a_{n+1}) \supset \ldots$$

and

$$\bigcap_{n=1}^{\infty} \Phi^{(n)}(a_1, \ldots, a_n) = a,$$

where $a = (a_1, \ldots, a_n, \ldots)$.

Such systems are usually called *converging systems.*

It can be proved [50] that in a converging number system, the set $\Phi^{(n)}(a_1, \ldots, a_n)$ is closed in the topology of real numbers, and therefore, $\sup \Phi^{(n)}(a_1, \ldots, a_n) = \max \Phi^{(n)}(a_1, \ldots, a_n)$.

Theorem 1. *If in a converging real number system (A, φ) with a finite alphabet A there is an online addition algorithm with delay δ, the system is fully redundant.*

Proof. Let us show that if there is an addition algorithm with an online delay δ, then for all $n \in \mathbb{N}$ there exists a string $(z_1, \ldots, z_n, \ldots) \in \varphi^{-1}(0)$ representing zero such that $\Phi^{(n)}(z_1, \ldots, z_n)$ contains a non-zero real number. Suppose, on the contrary, that there is an $n \in \mathbb{N}$ such that for all $(z_1, \ldots, z_n, \ldots) \in \varphi^{-1}(0)$, we have $\Phi^{(n)}(z_1, \ldots, z_n) = \{0\}$. Given an arbitrary real number a, let us denote the representation of $(-a)$ by $(a_1, \ldots, a_n, \ldots)$, and construct another number

$$a' = (a_1, \ldots, a_{n+\delta}, a'_1, a'_2, \ldots, a'_n, \ldots) \in R,$$

where $(a'_k)_{k\in\mathbb{N}} \in A^{\mathbb{N}}$ is any sequence of digits. The first n digits $(z_1, \ldots, z_n)$ of the online sum of a and a' are the first n digits of a string representing zero (the online sum of $(-a)$ and a), therefore according to our assumption, $a' = -a$. This means that our system can represent at most $(\operatorname{card} A)^{n+\delta}$ numbers, which is a contradiction because a finite subset of the reals cannot be dense in a neighbourhood of zero in $\mathbb{R}$.

Now let a be an arbitrary number represented by the sequence $(a_1, \ldots, a_n, \ldots)$, where $n \geq 1$ ($k = 1$ in Definition 5). Denote $A_n = \max \Phi^{(n)}(a_1, \ldots, a_n)$ and let us assume that $A_n < \max(\varphi(R))$. We have proved the existence of a string $z = (z_1, \ldots, z_n, \ldots) \in \varphi^{-1}(0)$ representing zero such that $\Phi^{(n+\delta)}(z_1, \ldots, z_{n+\delta})$ contains either a strictly positive or a strictly negative number[7]. Because our system is converging, we can find a strictly positive number $z^{(n)} = (z_1, \ldots, z_{n+\delta}, z'_{1n}, z'_{2n}, \ldots) > 0$ small enough for $A_n + z^{(n)}$ still to be representable, i.e. $A_n + z^{(n)} < \max(\varphi(R)))$. Since $z^{(n)} > 0$, the online sum $b = (b_1, \ldots, b_n, \ldots)$ of A_n and $z^{(n)}$ is strictly greater than A_n, and therefore the string $(b_1, \ldots, b_n)$ must be different from $(a_1, \ldots, a_n)$. But the online sum of A_n and z, whilst being another representation of A_n, also begins with the same prefix $(b_1, \ldots, b_n)$. We have now established that the number A_n is represented by *two* sequences with the distinct prefixes $(a_1, \ldots, a_n)$ and

[7]In fact, $\Phi^{(n+\delta)}(z_1, \ldots, z_{n+\delta})$ contains both positive *and* negative numbers.

$(b_1, \ldots, b_n)$ or, equivalently, that

$$A_n \in \Phi^{(n)}(a_1, \ldots, a_n) \cap \Phi^{(n)}(b_1, \ldots, b_n),$$

which is what had to be proved.

Note that if $A_n = \max \Phi^{(n)}(a_1, \ldots, a_n) = \max(\varphi(R))$, we could proceed symmetrically with $z^{(n)\prime} < 0$ and $A_n' = \min \Phi^{(n)}(a_1, \ldots, a_n)$. □

Redundant signed-digit expansions

As we have shown, adding redundancy in a number system is a necessary step to assure the existence of online algorithms on number representations. The following variant of the fixed-radix system was originally devised by Avizienis [5, 4] with the intention of defining an addition process free from carry propagation. Instead of the familiar set $\{1, 2, \ldots, (r-1)\}$, Avizienis used

$$\{-\rho, \ldots, -1, 0, 1, \ldots, \rho\},$$

where ρ is a positive number between $r/2$ and $r-1$.

For example, if the signed-digit set $\{-6, \ldots, -1, 0, 1, \ldots, 6\}$ is used to represent numbers in radix 10, then, say, 965 can be represented by $10\overline{4}5 = 1000 - 40 + 5$ (here $\overline{a} = -a$). The system is fully redundant, because the number of digits is greater than the radix, so that in the previous example, we could also write $965 = 1\overline{1}65 = 10\overline{4}5 = 10\overline{35}$.

Another example is the *signed binary system*, based upon radix 2 and the digit set $\{-1, 0, 1\}$.

Redundant signed-digit representations turn out to be very useful when developing algorithms for infinite precision arithmetic, and we shall use them very extensively in the chapters that follow.

Redundant continued fractions

It can be shown [87] that no algorithm can compute the first term of the regular continued fraction expansion of an arbitrarily given computable real number $x \in \mathbb{R}$ in finite time. This follows directly from Rice's Theorem that no function from $\mathbb{R}$ to a discrete set $\{0, 1\}$ is computable, unless it is *constant* [66]. Consequently, the regular continued fraction system cannot be used for exact real arithmetic.

Since continued fractions do not fall within the definition of a converging real number system as defined above due to the fact that the sets $\Phi^{(n)}(a_1, \ldots, a_n)$ are generally unbounded (and also because the alphabet $A = \mathbb{Z}$ is infinite), the results of Theorem 1 cannot be directly applied to the continued fraction system.

Nonetheless, continued fractions can also be made redundant and used for exact computations. A representation of computable real numbers by redundant continued fractions was proposed by Kornerup and Matula [46], and Vuillemin [87], who developed various incremental algorithms for basic arithmetic operations and for some transcendental functions, using the earlier work of Gosper [34].

1.6 An overview of previous work

One of the pioneer investigators of this problem was Wiedmer who was probably the first to suggest the use of redundant signed-digit systems to effect computability in exact computations [89]. His PhD thesis [90] contains an investigation of the algorithms necessary for exact real arithmetic on redundant signed-digit decimal sequences.

In 1981-2, Carl Pixley at Burroughs Corporation undertook a study of Wiedmer's work, implementing a package of functions for exact real arithmetic in the lazy functional language SASL. Pixley spent some time analyzing the efficiency of the algorithms, in particular for division, which is the most subtle of the four basic operations. Although never formally published, Pixley's work [62] was privately circulated and stimulated interest in the topic.

In 1986, Boehm, Cartwright, et al. [11] (also [12]) reported their two implementations of exact real arithmetic — as lazy infinite sequences of signed decimal digits, and as functions mapping rational errors to rational approximations. Having carried out a comparative study of the two methods, they contended that the lazy sequence method led to unsatisfactory implementations and performed very poorly, while the functional method performed surprisingly well. Their claim was partially based on what they called "the granularity effect" — computation of arguments to one digit's more accuracy than necessary, which makes the evaluation of expressions such as $x_1 + (x_2 + (x_3 + (\cdots + x_n)))$ highly inefficient. Boehm and Cartwright's functional approach was recently implemented by Valerie Ménissier-Morain [51]; yet, to this date no-one has attempted to reimplement the signed decimal (or radix-r for $r > 2$) digit method in order to find out whether the claimed advantage of functions over lazy lists of digits in a positional system was simply an artifact of a particular class of implementations of lazy languages, or evidence of something more fundamental.

Since 1980's, an extensive literature has arisen devoted to representations of exact reals, the most notable ones being the redundant continued fraction method developed by Vuillemin [87], and recently reimplemented by David Lester; Escardó's extension of PCF [28] based on infinite compositions of contracting affine maps $f : [0, 1] \to [0, 1]$ with rational coefficients, and a recent representation of exact real numbers based on the infinite composition of linear fractional transformations (infinite products of matrices) proposed by Edalat and Potts [26].

The only positional radix implementations that have appeared since Boehm and Cartwright are due to Pietro di Gianantonio [24, 25] who gives algorithms for the arithmetic operations in the signed binary system, and David Plume's exact real number calculator [63] based on the earlier work of Alex Simpson (see [75]) who also used the signed binary system. Gianantonio also investigated the unusual positional radix system based upon the golden ratio $\varphi = (\sqrt{5}+1)/2$, which requires only two digits 0 and 1 to represent numbers, while still being redundant. This author does not know of any existing positional radix implementation that uses a radix higher than 2.

Chapter 2

Exact real arithmetic in a radix-r system

2.1 Introduction and definitions

2.1.1 Radix-r redundant signed-digit expansions

A number system is said to be *redundant* if there are at least two distinct representations that are mapped onto the same number; otherwise, it is *non-redundant*. A radix r number system requires *at least* r digit symbols; if this number is greater than r, the system becomes redundant.

The following variation of the fixed-radix number system was originally used by Avizienis [5, 4] to eliminate carry propagation chains in addition and subtraction.

Definition 6. *A radix-r redundant signed-digit (SD) number system is one based on a digit set*

$$S_\rho = \left\{\overline{\rho}, \ldots, \overline{1}, 0, 1, \ldots, \rho\right\},$$

where $\overline{x}$ denotes $-x$, $1 \leq \rho \leq r-1$, and $\rho \geq r/2$.

The last condition allows each digit to assume more than r values and thus gives rise to the redundancy. We can measure the degree of redundancy of a given

SD system by calculating the *redundancy coefficient*

$$\mu(S_\rho) = \frac{\rho}{r-1}.$$

A digit set is said to be *maximally* or *minimally redundant* if its redundancy coefficient is maximal or minimal for the associated radix. Thus, for radix-10, the digit set $\left\{\overline{5}, \ldots, \overline{1}, 0, 1, \ldots, 5\right\}$ is minimally redundant, while $\left\{\overline{9}, \ldots, \overline{1}, 0, 1, \ldots, 9\right\}$ is maximally redundant. Henceforth, with the exception of Theorem 2, only non-minimally and non-maximally redundant systems will be considered (for why this is necessary, see Sections 2.3.1 and 2.5.1).

If $x \in \mathbb{R}$ is a real number, $r > 1$ an integer, and $(x_i)_{i \in \mathbb{N}_0}$ a sequence of integers with $-\rho \leq x_i \leq \rho$ for all $i \in \mathbb{N}$ such that

$$x = \sum_{i=0}^{\infty} x_i r^{-i},$$

then the symbol on the right side of

$$x = (x_0, x_1, x_2, \ldots, x_n, \ldots)_r \tag{7}$$

is called an *infinite radix-r redundant signed-digit expansion* for x. If $x_i = 0$ for all $i > p \geq 1$, we also write

$$x = (x_0, x_1, x_2, \ldots, x_p)_r$$

This is a *finite* or *terminating radix-r expansion* for x. In case $r = 10$ these are called decimal signed-digit expansions and the subscript 10 is omitted.

If we allow the first digit of signed-digit expansions to be unbounded ($x_0 \in \mathbb{Z}$), then every real number x has at least one radix-r redundant signed-digit expansion of the form (7). Some numbers have more than one, and sometimes even an infinite number of representations. How are all these expansions related to each other? In order to answer this question, we shall introduce a few concepts and definitions.

Definition 7. *Let $(a_n)_{n\in\mathbb{N}_0}$ be a sequence of integers such that the series*

$$\sum_{n=0}^{\infty} a_n r^{-n} \tag{8}$$

is convergent. A sequence $(b_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$ is said to be equivalent to $(a_n)_{n\in\mathbb{N}_0}$ if

$$\sum_{n=0}^{\infty} a_n r^{-n} = \sum_{n=0}^{\infty} b_n r^{-n}$$

(and, in particular, the series on the right is also convergent). To indicate the equivalence of two sequences, we shall use the symbol $\sim$.

If we denote by S the set of all sequences $(a_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$ for which the series (8) converges, then obviously $\sim$ is an equivalence relation on S, and using the fact that for any number $x \in \mathbb{R}$ there exists at least one expansion of the form (8), the equivalence classes are in one-to-one correspondence with the reals: $\mathbb{R} = S/\sim$.

We next define a family of functions $f : S \to S$ such that $f(s) \sim s$ for all $s \in S$.

Definition 8. *Let $i \in \mathbb{Z}$ be an integer, $i \neq 0$. We define*

$$f_0\left((a_n)_{n\in\mathbb{N}_0}\right) = (a_n)_{n\in\mathbb{N}_0}$$

$$f_i\left((a_n)_{n\in\mathbb{N}_0}\right) = (b_n)_{n\in\mathbb{N}_0}, \textit{ where } b_j = \begin{cases} a_j + \operatorname{sgn}(i), & \textit{if } j = |i| - 1 \\ a_j - \operatorname{sgn}(i)\cdot r, & \textit{if } j = |i| \\ a_j, & \textit{otherwise.} \end{cases}$$

Now let $(i_k)_{k=1}^{m}$ be a finite sequence of integers. We define

$$f_{i_1 i_2 \cdots i_m} \stackrel{def}{=} f_{i_m} \circ \ldots \circ f_{i_2} \circ f_{i_1}$$

For example, if $r = 10$, $(a_n)_{n\in\mathbb{N}_0} = (5, 5, \ldots 5, \ldots)$, we have

$$f_1(5, 5, 5, \ldots 5, \ldots) = (6, -5, 5, \ldots, 5, \ldots)$$
$$f_{-1}(5, 5, 5, \ldots 5, \ldots) = (4, 15, 5, \ldots, 5, \ldots)$$
$$f_{1,2}(5, 5, 5, \ldots 5, \ldots) = (6, -4, -5, 5, \ldots, 5, \ldots)$$
$$f_{-1,3}(5, 5, 5, \ldots 5, \ldots) = (4, 15, 6, -5, \ldots, 5, \ldots)$$

One can see that the n-th element of a sequence can only be changed by $f_{\pm n}$ and $f_{\pm(n+1)}$, so our next step is to carry over the definition of $f_{(i_k)}$ to the case where (i_k) is an infinite sequence.

Definition 9. *Let $(i_k)_{k\in\mathbb{N}}$, $i_k \neq 0$ be an unbounded sequence of integers such that the sequence $(|i_k|)_{k\in\mathbb{N}}$ is nondecreasing. We then define*

$$f_{(i_k)_{k\in\mathbb{N}}}\left((a_n)_{n\in\mathbb{N}_0}\right) \stackrel{def}{=} (b_n)_{n\in\mathbb{N}_0},$$

where $b_n = \left(f_{i_1\cdots i_{j_n}}\left((a_n)_{n\in\mathbb{N}_0}\right)\right)_n$ and $(j_n)_{n\in\mathbb{N}_0}$ is any sequence of natural numbers with $|i_{j_n-1}| \leq n < |i_{j_n}|$ (it is easy to verify that the value of b_n does not depend on the choice of $(j_n)_{n\in\mathbb{N}_0}$).

Since f_0 is the identity function, we can also allow zeros to appear in the sequence $(i_k)_{k\in\mathbb{N}}$ by agreeing to calculate the value of $f_{(i_k)_{k\in\mathbb{N}}}$ as

$$f_{(i_k)_{k\in\mathbb{N}}} \stackrel{def}{=} f_{(i'_k)_{k\in\mathbb{N}}}$$

where the sequence $(i'_k)_{k\in\mathbb{N}}$ is obtained from the original sequence $(i_k)_{k\in\mathbb{N}}$ by skipping all encountered zeros.

One of the main properties of the functions $f_{(i_k)}$ is that they do not take us out of the equivalence classes with respect to $\sim$, i.e. for any $(a_n)_{n\in\mathbb{N}_0} \in S$

$$f_{i_1 i_2\cdots i_m}\left((a_n)_{n\in\mathbb{N}_0}\right) \sim (a_n)_{n\in\mathbb{N}_0}$$
$$f_{(i_k)_{k\in\mathbb{N}}}\left((a_n)_{n\in\mathbb{N}_0}\right) \sim (a_n)_{n\in\mathbb{N}_0}$$

Among other properties, we can indicate that

$$f_{m,-m} = f_{-m,m} = f_0 \text{ for all } m \in \mathbb{N}$$

Theorem 2. *Let $r \in \mathbb{N}$, $r > 1$ be a radix value, ρ be an integer with $1 \le r/2 \le \rho \le r-1$, and let $x \in \mathbb{R}$. Then there exists a sequence $(a_n)_{n\in\mathbb{N}_0}$ of integers such that $-\rho \le a_n \le \rho$ for all $n \in \mathbb{N}$, and*

$$x = \sum_{n=0}^{\infty} a_n r^{-n}$$

Moreover, if $(b_n)_{n\in\mathbb{N}_0}$ is any other ($a_j \ne b_j$ for some j) sequence of integers such that $-\rho \le b_n \le \rho$ for all $n \in \mathbb{N}$ (if $\rho = r-1$, we also require that $b_n \ne r-1$ for infinitely many n, and $b_n \ne -r+1$ for infinitely many n), and

$$x = \sum_{n=0}^{\infty} b_n r^{-n}, \tag{9}$$

then there exists a (possibly finite) integer sequence $(i_k)_{k\in\mathbb{N}}$ such that $(|i_k|)_{k\in\mathbb{N}}$ is nondecreasing and $(b_n)_{n\in\mathbb{N}_0} = f_{(i_k)_{k\in\mathbb{N}}}\left((a_n)_{n\in\mathbb{N}_0}\right)$.

Proof. Let $x_0.x_1x_2\ldots x_n\ldots$ be the conventional radix-r expansion of x, i.e. $x_0 \in \mathbb{Z}$, $0 \le x_i < r$ for all $i \in \mathbb{N}$. Then we define

$$(a_n)_{n\in\mathbb{N}_0} = f_{(i_n)_{n\in\mathbb{N}}}\left((x_n)_{n\in\mathbb{N}_0}\right)$$

where

$$(i_n)_{n\in\mathbb{N}} = \begin{cases} n, & \text{if } \rho \le x_n \le r-1 \\ 0, & \text{if } 0 \le x_n \le \rho - 1 \end{cases}$$

It is quite easy to see that $|a_n| \le \rho$ for all $n \in \mathbb{N}$: we know that $0 \le x_n < r$, and a_n is obtained from x_n through application of $f_{i_1\ldots i_k}$ for some $i_1, \ldots i_k$. Thus, if $x_n \in [0, \rho-1]$, it may only be changed by f_{n+1}, in which case it will be increased by 1; if $x_n \in [\rho, r-1]$, then f_n will reduce its value by r, and the resulting value may, in its turn, be also increased by 1 by f_{n+1}. In either case, we have $-\rho \le a_n \le \rho$, and $x = \sum_{n=0}^{\infty} a_n r^{-n}$.

Now suppose that (9) obtains for some sequence $(b_n)_{n\in\mathbb{N}_0}$ of integers where $|b_n| \leq \rho$ for all $n \in \mathbb{N}$, and $b_j \neq a_j$ for some j. Let $k = \inf\{j \in \mathbb{N} : a_j \neq b_j\}$, then we have

$$\sum_{n=k}^{\infty} a_n r^{-n} = \sum_{n=k}^{\infty} b_n r^{-n}$$

or

$$b_k = a_k + \sum_{n=1}^{\infty} (a_{n+k} - b_{n+k})\, r^{-n}$$

Since $|a_{n+k}| \leq \rho$, $|b_{n+k}| \leq \rho$, we can estimate

$$\left|\sum_{n=1}^{\infty} (a_{n+k} - b_{n+k})\, r^{-n}\right| \leq \sum_{n=1}^{\infty} |a_{n+k} - b_{n+k}|\, r^{-n} \leq \sum_{n=1}^{\infty} 2\rho r^{-n} = \frac{2\rho}{r-1}$$

Generally, we have $1 < 2\rho/(r-1) \leq 2$, but the pathological equality $b_k = a_k \pm 2$ may only hold true in the case where $\rho = r-1$ and $x = (a_0, a_1, \ldots, a_k, \pm\rho, \pm\rho, \pm\rho \ldots) = (b_0, b_1, \ldots, b_k, \mp\rho, \mp\rho, \mp\rho, \ldots)$, which we have excluded from consideration. Hence, we deduce that

$$b_k = a_k \pm 1$$

Now we set

$$i_1 = \begin{cases} k, & \text{if} \quad b_k = a_k + 1 \\ -k, & \text{if} \quad b_k = a_k - 1 \end{cases}$$

and if $(a'_n)_{n\in\mathbb{N}_0} = f_{i_1}\left((a_n)_{n\in\mathbb{N}_0}\right)$, then $b_n = a'_n$, $n = \{1, \ldots, k\}$.

Once $i_1, \ldots, i_{n-1}$ have been chosen, let $i_n = |i_n| \cdot \operatorname{sgn}(i_n)$, where

$$|i_n| = \inf\left\{j \in \mathbb{N}_0 : b_j \neq \left(f_{i_1 \ldots i_{n-1}}\left((a_n)_{n\in\mathbb{N}_0}\right)\right)_j\right\}$$

$$\operatorname{sgn}(i_n) = \begin{cases} 1, & \text{if} \quad b_{|i_n|} = \left(f_{i_1 \ldots i_{n-1}}\left((a_n)_{n\in\mathbb{N}_0}\right)\right)_{|i_n|} + 1 \\ -1, & \text{if} \quad b_{|i_n|} = \left(f_{i_1 \ldots i_{n-1}}\left((a_n)_{n\in\mathbb{N}_0}\right)\right)_{|i_n|} - 1 \end{cases}$$

It may happen that $i_n = 0$ for all $n > p$, $p \in \mathbb{N}$. In this case, we shall consider the resulting sequence $(i_1, \ldots, i_p)$ to be finite. This concludes the proof. $\square$

2.2 The representation

We aim to represent real numbers by sequences from the representation set S, as defined in Section 2.1.1. For example, one might define a computable exact real number x as a triple (r, E, M), where $E \in \mathbb{Z}$ is an exponent, M is a mantissa which is a sequence of numbers $(a_n)_{n\in\mathbb{N}_0} \in S$, and the value of x is computed as

$$x = r^E \cdot \sum_{n=0}^{\infty} a_n r^{-n}. \tag{10}$$

Such a representation, however, would be too loose a concept to be useful by itself. We must also provide some constructive condition in order to guarantee convergence of the series in (10) and be able to make useful inferences about a number from a finite amount of information about its representation.

Definition 10. *We define a representation of an exact real number x to be a quadruple (r, ρ, E, M), where $r \in \mathbb{N}$, $r > 2$ is the radix value, the range parameter ρ is an integer with $r/2 < \rho < r-1$, $E \in \mathbb{Z}$ is a signed exponent, M is a mantissa, which is an effectively given[1] sequence of numbers $(a_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$ such that*

$$|a_n| \leq Cn,\ n \in \mathbb{N}, \tag{11}$$

where $C > 0$ is a constant, common to all real numbers in a given system — we therefore do not include it in the representation[2]. The representation $(r, \rho, E, (a_n)_{n\in\mathbb{N}_0})$ is said to be canonical or normalized, if

$$|a_n| \leq \rho,\ n \in \mathbb{N}_0.$$

[1] *The sequence $(a_n)_{n\in\mathbb{N}_0}$ could in principle be given by an oracle — it does not necessarily have to be computable in the sense of being the sequence of values $g(0), g(1), g(2), \ldots$ of a general recursive function $g(x)$.*

[2] *The convergence criterion (11) is somewhat arbitrary and only required to ensure effective convergence of the sequence. If a sequence were found to violate (11), an error message would be produced at run-time. The value of $C = 2(r-1)^2$ could be given as a rough estimate that satisfies the algorithmic requirements.*

The value of $x = (r, \rho, E, (a_n)_{n\in\mathbb{N}_0})$ is taken as in (10). Later on we will centre on the factors that affect the choice of appropriate values for the parameters r and ρ.

For brevity and ease of reading, we shall not always distinguish between a number x and its representation (r, ρ, E, M), and refer to a number as normalized if its representation is normalized, and vice versa. We shall also assume that r and ρ are fixed and sometimes use the notation (E, M) instead of (r, ρ, E, M).

Observe that we can view a *finite* number as being infinite, by attaching an infinite sequence of zeros at the end of its mantissa:

$$r^E \cdot \sum_{i=0}^{N} a_i r^{-i} = r^E \cdot \sum_{i=0}^{\infty} a_i r^{-i},$$

where we have set $a_i = 0$ for $i > N$. We can therefore assume, without any restriction of generality, that the mantissas of all operands are infinite, unless otherwise specified.

The requirement $\rho < r - 1$ excludes maximally redundant systems from consideration, which brings to pass the following nice feature of the canonical $-\rho$-to-ρ representation: the sign of a number is determined by the sign of the first non-zero entry of its mantissa (see Section 2.5.1).

2.3 Normalization

2.3.1 Normalization of bounded infinite strings

Most algorithms presented in this and subsequent chapters assume that all operands are normalized, and also require normalization of the results, so we shall now discuss the algorithms for normalizing real numbers. We recall that normalization refers to the process of restoring the individual digits of a real number's mantissa $(a_i)_{i\in\mathbb{N}_0}$ to the canonical range $[-\rho, \rho]$.

Let $(a_i)_{i\in\mathbb{N}_0}$ be an unnormalized mantissa of a real number $a = r^E \cdot \sum_{i=0}^{\infty} a_i r^{-i}$.

We shall first confine our attention to the case where $|a_i| \le r+\rho-1,\ i \in \mathbb{N}_0$, and show how to obtain a new exponent E' and mantissa $(a'_i)_{i\in\mathbb{N}_0}$ such that

$$\begin{aligned} &1) \quad a = r^E \cdot \sum_{i=0}^{\infty} a_i r^{-i} = r^{E'} \cdot \sum_{i=0}^{\infty} a'_i r^{-i} \\ &2) \quad |a'_i| \le \rho,\ i \in \mathbb{N}_0 \end{aligned} \tag{12}$$

To this end, we first consider $\sum_{i=0}^{\infty} a_i r^{-i}$ and repeatedly divide a_i by r for all $i \in \mathbb{N}_0$:

$$a_i = d_i r + m_i, \quad |m_i| < r, \quad \operatorname{sgn}(m_i) = \operatorname{sgn}(d_i) \tag{13}$$

We have:

$$\begin{aligned} \sum_{i=0}^{\infty} a_i r^{-i} &= \sum_{i=0}^{\infty} (d_i r + m_i)\, r^{-i} = \sum_{i=0}^{\infty} d_i r^{-i+1} + \sum_{i=0}^{\infty} m_i r^{i} \\ &= (d_0 r + m_0 + d_1) + \sum_{i=1}^{\infty} (m_i + d_{i+1})\, r^{-i} \end{aligned} \tag{14}$$

Now $|a_i| \le r+\rho-1$ implies $|d_i| \le 1$, $|m_i| \le r-1$ $(i \in \mathbb{N}_0)$ and, thus, $|m_i + d_{i+1}| \le r$. We, however, aim to obtain a value less or equal to ρ (instead of r). Let us introduce the following notation:

$$\begin{aligned} d'_i &= \begin{cases} d_i, & \text{if } |m_i| < \rho \\ d_i + \operatorname{sgn}(m_i), & \text{if } |m_i| \ge \rho \end{cases} \\ m'_i &= \begin{cases} m_i, & \text{if } |m_i| < \rho \\ m_i - \operatorname{sgn}(m_i)\cdot r, & \text{if } |m_i| \ge \rho \end{cases} \\ a'_i &= m'_i + d'_{i+1}\ (i \in \mathbb{N}), \quad a'_0 = a_0 + d'_1 \end{aligned} \tag{15}$$

From (15) it can be seen that $a_i = d'_i r + m'_i$. and, similarly to (14), we arrive at

$$\sum_{i=0}^{\infty} a_i r^{-i} = (a_0 + d'_1) + \sum_{i=1}^{\infty} \left(m'_i + d'_{i+1}\right) r^{-i} = \sum_{i=0}^{\infty} a'_i r^{-i}$$

Let us now verify that $|a'_i| \le \rho$ for all $i \in \mathbb{N}$. For this purpose we note that $|d'_i| \le 1$

for all $i \in \mathbb{N}_0$, so if $|m_i| < \rho$, then $m_i' = m_i$, $|m_i'| < \rho$, and $|a_i'| = \left|m_i' + d_{i+1}'\right| \leq \rho$. If $|m_i| \geq \rho$, then $m_i' = m_i - \operatorname{sgn}(m_i) \cdot r$, $1 \leq |m_i'| \leq r - \rho$, and $|a_i'| = \left|m_i' + d_{i+1}'\right| \leq r - \rho + 1$. Now we require $r - \rho + 1 \leq \rho$ or, equivalently,

$$\rho \geq \frac{r+1}{2} \tag{16}$$

which in its turn implies that $(r+1)/2 \leq r - 1$ or $r \geq 3$. Using (16), we finally obtain $|a_i'| \leq \rho$ for all $i \in \mathbb{N}$. For $i = 0$, however, the inequality does not necessarily hold true. On the other hand, from the definition of a_0' we can conclude that $a_0 - 1 \leq a_0' \leq a_0 + 1$.

In this manner, we have constructed a function $f : \mathbb{N}^2 \times \mathbb{Z}^{\mathbb{N}_0} \to \mathbb{Z}^{\mathbb{N}_0}$ (it will be referred to as `reduce`) which assigns to any triple $\left(r, \rho, (a_i)_{i \in \mathbb{N}_0}\right)$ the sequence $(a_i')_{i \in \mathbb{N}_0} \in \mathbb{Z}$, calculated according to formulae (15). Evaluation of this function can be performed totally in parallel (Figure 1), and in practice the steps (13) and (15) are performed in a single step.

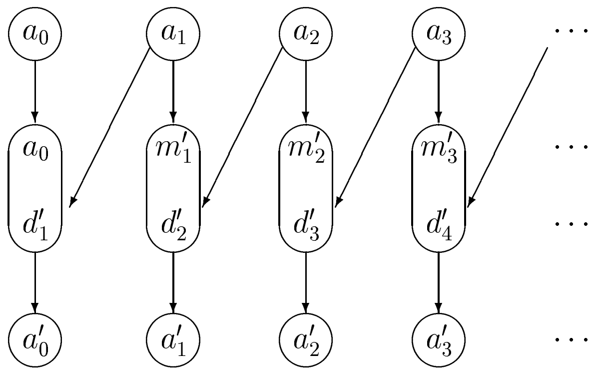


Figure 1: Totally parallel normalization

Returning to formula (12), we now construct the promised number as follows:

$$E' = \begin{cases} E, & \text{if } |a_0| \leq \rho - 1 \\ E + 1, & \text{if } |a_0| \geq \rho \end{cases}$$

$$(a_i')_{i \in \mathbb{N}_0} = \begin{cases} f\left((a_i)_{i \in \mathbb{N}_0}\right), & \text{if } |a_0| \leq \rho - 1 \\ f\left((0, a_0, a_1, \ldots, a_n, \ldots)\right), & \text{if } |a_0| \geq \rho \end{cases}$$

Now let us consider a more general case where $|a_i| \leq M,\ i \in \mathbb{N}_0$, where $M > 0$

is an arbitrary positive integer. We can now easily show that it is possible to normalize mantissa $(a_i)_{i\in\mathbb{N}_0}$ in a finite number of steps. Indeed, applying `reduce`, we shall obtain a sequence $(a'_i)_{i\in\mathbb{N}_0}$, satisfying the following condition:

$$|a'_i| = \left|m'_i + d'_{i+1}\right| \leq |m'_i| + \left|d'_{i+1}\right| \leq \rho - 1 + \left\lfloor \frac{M}{r} \right\rfloor + 1,$$

or

$$|a'_i| \leq M_1 \stackrel{def}{=} \left\lfloor \frac{M}{r} \right\rfloor + \rho.$$

Applying `reduce` again, we get another sequence $(a''_i)_{i\in\mathbb{N}_0}$, satisfying

$$|a''_i| \leq M_2 \stackrel{def}{=} \left\lfloor \frac{M_1}{r} \right\rfloor + \rho,$$

etc. The sequence $M, M_1, M_2, \ldots$ is a sequence of decreasing natural numbers, and if $M = m_n r^n + \ldots + m_1 r + m_0$, the algorithm will terminate in at most $n+1$ steps.

More specifically, we can prove the following result.

Theorem 3. *Let $(a_i)_{i\in\mathbb{N}_0}$ be a sequence with $|a_i| \leq M$, $i \in \mathbb{N}$, where M is an arbitrary positive integral number. In order that the sequence $(a_i)_{i\in\mathbb{N}_0}$ be normalized to an equivalent sequence $(b_i)_{i\in\mathbb{N}_0}$ with $|b_i| \leq N$, $i \in \mathbb{N}$ on applying* `reduce` *at most n times, it is sufficient that $M \leq g^{(n)}(N)$, where $g^{(n)}(x) = r^n x + C_n$, $C_n = C\left(r^{n-1} + \ldots + r + 1\right)$, $C = (r-1)(1-\rho)$.*

Proof. To prove the sufficiency of the condition imposed upon M, we need but note that the functions $g^{(n)}(x)$ satisfy the following recurrence formulae

$$g^{(n)}(x) = r \cdot g^{(n-1)}(x) + C,\ n \in \mathbb{N},$$

where $g^{(0)}(x) \equiv x$. Equivalently,

$$g^{(n)}(x) = \underbrace{g \circ g \circ \cdots \circ g}_{n \text{ times}}(x),$$

where

$$g(x) = g^{(1)}(x) = rx + C.$$

Thus, it suffices to show that any sequence $(a_i)_{i \in \mathbb{N}_0}$ with $|a_i| \le g(x)$, $i \in \mathbb{N}$ can be reduced, in a single step, to a sequence $(a'_i)_{i \in \mathbb{N}_0}$ with $|a'_i| \le x$, $i \in \mathbb{N}$.

Let $(a_i)_{i \in \mathbb{N}_0}$ be any such sequence, i.e. $|a_i| \le rx + C_1$, $i \in \mathbb{N}$. As indicated above, $|a_i| \le M$ implies $|a'_i| \le M_1 = \left\lfloor \frac{M}{r} \right\rfloor + \rho$, and thus picking $M = rx + C_1$ yields $M_1 = \left\lfloor x + (1-\rho)\frac{r-1}{r} \right\rfloor + \rho < x + (1-\rho) + \rho = x + 1$, i.e.

$$|a'_i| \le x,$$

which is what had to be proved. □

Corollary 1. *If $(a_i)_{i \in \mathbb{N}_0}$ is a sequence satisfying $|a_i| \le g^{(n)}(\rho)$ for some $n \in \mathbb{N}$ and all $i \in \mathbb{N}$, it can be fully normalized by `reduce` in at most n steps.*

This follows immediately from the theorem: n normalizations give us a sequence $(a'_i)_{i \in \mathbb{N}_0}$ with $|a'_i| \le \rho$.

The converse statement is not necessarily true: even if $|a_k| > g^{(n)}(N)$ for some $k \in \mathbb{N}$, after n normalizations we may still get a $(b_i)_{i \in \mathbb{N}_0}$ with $|b_i| \le N$ for all $i \in \mathbb{N}$. Suppose, for instance, that $a_k = g(N) + 1 = r(N - \rho + 1) + \rho$ for some $k \in \mathbb{N}$ and $|a_i| \le g(N)$ for $i \ne k$. This implies that $d_k = N - \rho + 1$, $m_k = \rho$ and, therefore, $d'_k = N - \rho + 2$, $m'_k = \rho - r$. Recalling that

$$a'_{k-1} = m'_{k-1} + d'_k, \; k \in \mathbb{N},$$

one can see that the larger-than-usual value of d'_k can only affect the $(k-1)$-st element of the resulting sequence, and, further still, only if $m'_{k-1} = \rho - 1$, in which case $a'_{k-1} = \rho - 1 + N - \rho + 2 = N + 1$. However, the value of m'_{k-1} depends solely on a_{k-1}, and can be anywhere in the range from $-\rho + 1$ to $\rho - 1$, irrespective of the value of the next element, a_k. If it so happens that $m'_{k-1} \ne \rho - 1$, we will have $\left|a'_{k-1}\right| \le N$, and consequently — since the a'_i for $i \ne k-1$ have remained intact — $|a'_i| \le N$ for all $i \in \mathbb{N}$.

This example shows that the functions $g^{(n)}(x)$ give us, in fact, the best upper bound one could possibly have in order that *any* sequence bounded by it be safely normalized. More precisely, for any integer function $f^{(n)}(x) > g^{(n)}(x)$ there is a sequence $(a_i)_{i\in\mathbb{N}_0}$ with $|a_i| \leq f^{(n)}(\rho)$ that cannot be fully normalized in n applications of `reduce`.

By way of illustration, let us give a few examples.

Example 1. *Let $r = 6$, $\rho = 4$, $(a_i)_{i\in\mathbb{N}_0}$ be a sequence with $|a_i| \leq 3500$ for all $i \in \mathbb{N}$. How many times does one have to apply `reduce` to obtain an equivalent sequence $(a_i')_{i\in\mathbb{N}_0}$ with $|a_i'| \leq 100$, $i \in \mathbb{N}$? We have*

$$\begin{aligned} g(100) &= 6 \cdot 100 - 15 = 585 \\ g^{(2)}(100) &= g(585) = 6 \cdot 585 - 15 = 3495 \\ g^{(3)}(100) &= g(3495) = 6 \cdot 3495 - 15 = 20955 \end{aligned}$$

Since $g(100) < g^{(2)}(100) < 3500 < g^{(3)}(100)$, 3 normalizations will be sufficient by Theorem 3.

Example 2. *Let $r = 10$, $\rho = 6$. Find the bound for the elements of a sequence that can be fully normalized in 3 applications of `reduce`. According to Corollary 1, we need but calculate*

$$g^{(3)}(6) = 1000 \cdot 6 + 999 \cdot (-5) = 1005.$$

Thus, if $|a_i| \leq 1005$, $i \in \mathbb{N}$, $(a_i)_{i\in\mathbb{N}_0}$ can be fully `reduce`d in three passes.

The functions $g^{(n)}(x)$ have a much simpler form when $x = \rho$: indeed, it is easy to see that

$$g^{(n)}(\rho) = r^n + \rho - 1,\ n \in \mathbb{N}. \tag{17}$$

The right-hand side of equality (17) is solvable for n, which enables us to determine the number of times one has to apply `reduce` in order to *fully* normalize a given sequence $(a_i)_{i\in\mathbb{N}_0}$. In more exact terms, let $(a_i)_{i\in\mathbb{N}_0}$ be a sequence with

$|a_i| \le M$, $i \in \mathbb{N}$. By theorem 3,

$$n = \min\left\{k \in \mathbb{N} \;\middle|\; M \le g^{(k)}(\rho)\right\}.$$

Solving the inequality $M \le g^{(k)}(\rho)$ for $k \in \mathbb{N}$, we find that

$$k \ge \log_r (M - \rho + 1),$$

or,

$$n = \lceil \log_r (M - \rho + 1) \rceil \tag{18}$$

As a conclusion, let us take note of the fact that, as it follows from (16), in order for our system to allow totally parallel normalization, i.e. absence of carry propagation chains, it must *not* be minimally redundant. For $r = 2$, for instance, there is only one possible digit set, $\{\bar{1}, 0, 1\}$; thus, in the binary case the condition $\rho \ge (r+1)/2 = 3/2$ cannot be satisfied.

2.3.2 Normalization of unbounded strings

Let $(a_n)_{n\in\mathbb{N}_0}$ be an unnormalized mantissa of a real number $a = r^E \cdot \sum_{n=0}^{\infty} a_n r^{-n}$ such that $|a_n|$ are unbounded. It is clear that some knowledge of the behaviour of a_n is needed before normalization can proceed. In what follows, we specify a way to establish the growth rate of a sequence of numbers, and then show how to normalize sequences using information thus obtained.

Let $(n_k)_{k\in\mathbb{N}_0}$ be an arbitrary computable sequence of natural numbers. For any such sequence, and for any given sequence $(a_n)_{n\in\mathbb{N}_0}$, we can find another computable sequence of numbers $(M_k)_{k\in\mathbb{N}_0}$ such that

$$|a_n| \le \sum_{i=0}^{k} M_i, \quad \text{if } n < \sum_{i=0}^{k} n_i. \tag{19}$$

In other words, the first n_0 terms of sequence $(a_n)_{n\in\mathbb{N}_0}$ are bounded by M_0, the next n_1 terms bounded by $M_0 + M_1$, and so on:

$$\underbrace{a_0 \;\cdots\; a_{n_0-1}}_{\leq M_0} \; \underbrace{a_{n_0} \cdots \; a_{n_0+n_1-1}}_{\leq M_0+M_1} \; \underbrace{a_{n_0+n_1} \cdots \; a_{n_0+n_1+n_2-1}}_{\leq M_0+M_1+M_2} \cdots \tag{20}$$

It is easy to see how such $(M_k)_{k\in\mathbb{N}_0}$ can be constructed, e.g. we can take

$$\begin{aligned}
M_0 &= \max\left\{|a_0|, \ldots, |a_{n_0-1}|\right\}, \\
M_1 &= \max\left\{|a_{n_0}|, \ldots, |a_{n_0+n_1-1}|\right\} - M_0, \\
M_2 &= \max\left\{|a_{n_0+n_1}|, \ldots, |a_{n_0+n_1+n_2-1}|\right\} - M_0 - M_1, \\
&\ldots
\end{aligned}$$

Given (19), we can split our sequence $(a_n)_{n\in\mathbb{N}_0}$ into an infinite number of sequences $(a_{kn})_{n\in\mathbb{N}_0}$, $k \in \mathbb{N}_0$, each of which is bounded by M_k, and whose sum gives us the original sequence:

$$\begin{array}{llllllllll}
\leq M_0 & a_{00} & \cdots & a_{0,n_0-1} & a_{0,n_0} & \cdots & a_{0,n_0+n_1-1} & a_{0,n_0+n_1} & \cdots & a_{0,n_0+n_1+n_2-1} & \cdots \\
\leq M_1 & & & & a_{1,0} & \cdots & a_{1,n_1-1} & a_{1,n_1} & \cdots & a_{1,n_1+n_2-1} & \cdots \\
\leq M_2 & & & & & & & a_{2,0} & \cdots & a_{2,n_2-1} & \cdots \\
\ldots & & & & & & & & & & \ddots \\
 & a_0 & \cdots & a_{n_0-1} & a_{n_0} & \cdots & a_{n_0+n_1-1} & a_{n_0+n_1} & \cdots & a_{n_0+n_1+n_2-1} & \cdots
\end{array}$$

In exact terms, we split (20) into multiple sequences $(a_{kn})_{n\in\mathbb{N}_0}$ such that $|a_{kn}| \leq M_k$ and

$$a_n = \begin{cases}
a_{0n}, & \text{if} \quad 0 \leq n < n_0 \\
a_{0,n} + a_{1,n-n_0} & \text{if} \quad n_0 \leq n < n_0 + n_1 \\
a_{0,n} + a_{1,n-n_0} + a_{2,n-n_0-n_1} & \text{if} \quad n_0 + n_1 \leq n < n_0 + n_1 + n_2 \\
\quad \cdots & \quad \cdots
\end{cases}$$

Now each of the sequences $(a_{kn})_{n\in\mathbb{N}_0}$ is bounded and therefore can be normalized by the usual reduction procedure in $\lceil \log_r (M_k - \rho + 1) \rceil$ steps, according to (18). To normalize the original sequence $(a_n)_{n\in\mathbb{N}_0}$, we start by normalizing $(a_{0n})_{n\in\mathbb{N}_0}$ in $\lceil\log_r(M_0-\rho+1)\rceil$ applications of `reduce`, which gives us n_0 digits of normalized result. We then proceed to add the next sequence $(a_{1n})_{n\in\mathbb{N}_0}$ and renormalize the result $\lceil\log_r(M_1+1)\rceil$ times (the sum of the two sequences is bounded by $M_1+\rho$), which gives us the next n_1 digits of the result, and so on (processing sequence (a_{kn}) gives us additional n_k digits).

Note that this algorithm cannot normalize an arbitrarily given unbounded sequence $(a_n)_{n\in\mathbb{N}_0}$, unless its growth rate sequence $(M_k)_{k\in\mathbb{N}_0}$ is known in advance. In particular, the algorithm is unable to detect a divergent sequence, and if given one, would attempt to normalize it, although of course it will fail and the resulting sequence will still be divergent.

We shall review this procedure in great detail in Section 2.4.2 where we discuss multiplication, which produces unbounded unnormalized sequences of digits. We shall also elaborate upon the choice of n_k and the resulting complexity in Section 4.1.3 of Chapter 4.

2.3.3 Partial normalization

Normalization of an infinite sequence of digits is a computationally intensive procedure, and in some algorithms it is possible to avoid full normalization, applying partial normalization instead. For instance, a typical division algorithm involves the computation of a new remainder for each quotient digit, and as we show in Section 2.4.3, only a finite number of the most significant digits of each remainder need actually be normalized at each step.

In this section, we introduce a new function, `freduce`, which is designed to be a substitute for `reduce` in situations where partial normalization can be employed. `freduce` takes one extra parameter, precision $n\in\mathbb{N}$, and normalizes only the first n elements of a sequence. As noted earlier (Section 2.2), a finite number can be viewed as an infinite one if we append a list of zeros to its mantissa; hence, `reduce` can operate on finite sequences as well as infinite ones. As can be seen from (13) and (15), if we apply `reduce` to a finite sequence

$$a = (a_0, a_1, \ldots, a_n) = (a_0, a_1, \ldots, a_n, 0, \ldots, 0, \ldots),$$

the resulting sequence $a' = (a'_0, a'_1, \ldots, a'_n, a'_{n+1}, \ldots)$ will have elements $a'_{n+1} = a'_{n+2} = \ldots = 0$ and can be considered finite. Therefore, we can define

$$\texttt{freduce}\ (n, (a_k)_{k\in\mathbb{N}_0}) = (a'_0, a'_1, \ldots, a'_n, a_{n+1}, a_{n+2}, \ldots),$$

where $(a'_0, a'_1, \ldots, a'_n) = \texttt{reduce}\ (a_0, a_1, \ldots, a_n)$.

2.4 Basic arithmetic operations

2.4.1 Addition and subtraction

In this section, we shall discuss algorithms for the operations of exact real addition and subtraction. The emphasis will mainly be on the former, since subtraction is usually carried out as the addition of a negated number. We shall first discuss addition of two numbers and then look at multiple number addition.

Addition of two numbers

Let $a = r^{E_a} \cdot \sum_{i=0}^{\infty} a_i r^{-i}$ and $b = r^{E_b} \cdot \sum_{j=0}^{\infty} b_j r^{-j}$ be the two normalized radix-r numbers to be added. Since the addition operation is commutative, we can assume $e = E_a - E_b \geq 0$ without loss of generality. The procedure for addition or subtraction is as follows:

$$\begin{aligned} a + b &= r^{E_a} \cdot \sum_{i=0}^{\infty} a_i r^{-i} + r^{E_b} \cdot \sum_{i=0}^{\infty} b_i r^{-i} = r^{E_a} \left(\sum_{i=0}^{\infty} a_i r^{-i} + r^{-e} \cdot \sum_{i=0}^{\infty} b_i r^{-i} \right) \\ &= r^{E_a} \cdot \sum_{i=0}^{\infty} \left(a_i + b'_i\right) r^{-i}, \end{aligned}$$

where

$$(b'_0, b'_1, b'_2, \cdots, b'_n, \cdots)_r = \left(\underbrace{0, 0, \cdots, 0}_{e \text{ zeros}}, b_0, b_1, b_2, \cdots \right)_r .$$

Thus, in order to perform addition, we must first adjust the mantissa of one of the operands to make the two exponents equal (align the radix points), and then add the two sequences digit by digit. The resulting sequence

$$(a_0 + b'_0,\ a_1 + b'_1, \cdots,\ a_n + b'_n,\ \cdots)_r$$

can then be normalized in a single pass, since

$$|a_n + b'_n| \leq |a_n| + |b'_n| \leq 2\rho \leq r + \rho - 1.$$

Addition of several numbers

The above addition algorithm can be readily modified to operate with k numbers, where $k > 2$. The procedure is essentially the same — the mantissas of all k numbers are first aligned to match the one with the largest exponent, and then added digit-by-digit. As it follows from (18), the resulting sequence can be normalized by applying `reduce` $\lceil \log_r (k\rho - \rho + 1) \rceil$ times.

Note that this is considerably more efficient than adding the k numbers pairwise using $(k - 1)$ nested additions, as we discuss later (Section 4.1.2).

Subtraction

Subtraction is carried out in the usual way by negating the minuend and adding the result to the subtrahend. Negation is performed as follows:

$$-\sum_{i=0}^{\infty} a_i r^{-i} = \sum_{i=0}^{\infty} (-a_i) \cdot r^{-i}$$

2.4.2 Multiplication

Let the multiplier and multiplicand be denoted by $a, b \in \mathbb{R}$ respectively, with the following normalized sequences of signed digits:

$$(a_0, a_1, a_2, \cdots, a_n, \cdots), \quad (b_0, b_1, b_2, \cdots, b_n, \cdots),$$

i.e.

$$a = r^{E_a} \cdot \sum_{i=0}^{\infty} a_i r^{-i}, \quad b = r^{E_b} \cdot \sum_{j=0}^{\infty} b_j r^{-j}$$

Then

$$ab = r^{E_a+E_b} \cdot \left(\sum_{i=0}^{\infty} a_i r^{-i} \right) \cdot \left(\sum_{j=0}^{\infty} b_j r^{-j} \right)$$

The Cauchy product of the two series $\sum_{i=0}^{\infty} a_i r^{-i}$ and $\sum_{j=0}^{\infty} b_j r^{-j}$ is the series

$$\sum_{m=0}^{\infty} c_m r^{-m} = \sum_{m=0}^{\infty} \left(\sum_{i=0}^{m} a_i b_{m-i} \right) r^{-m},$$

where $c_m = \left(\sum_{i=0}^{m} a_i b_{m-i} \right)$. Since both series $a = \sum_{n=0}^{\infty} a_n r^{-n}$ and $b = \sum_{n=0}^{\infty} b_n r^{-n}$ are absolutely convergent, by Mertens' theorem (see e.g. [80]) their Cauchy product $\sum_{n=0}^{\infty} c_n r^{-n}$ converges to ab.

Since $(a_i)_{i\in\mathbb{N}_0}$ and $(b_i)_{i\in\mathbb{N}_0}$ are canonical representations of a and b, we have

$$|c_m| \leq \rho^2 \cdot (m+1)\,, \; m \in \mathbb{N}_0\,.$$

Now we want to find the result in the form

$$ab = c = r^{E_c} \cdot \sum_{m=0}^{\infty} c'_m r^{-m},$$

where $-\rho \leq c'_m \leq \rho$ for all $m \in \mathbb{N}_0$. The sequence $(c_m)_{m\in\mathbb{N}_0}$ generally is not bounded by any positive integer, therefore we have to use the unbounded normalization method described in Section 2.3.2. To do so, we shall use the sequence

$n_k = n$ for some $n \in \mathbb{N}$ (see Section 2.3.2) and recursively apply `reduce` to small bounded portions of $(c_m)_{m \in \mathbb{N}_0}$, as shown in Figures 2 and 3.

$$
\begin{array}{cccc|ccccc|c}
a_0b_0 & a_0b_1 & \cdots & a_0b_{n-1} & a_0b_n & a_0b_{n+1} & a_0b_{n+2} & \cdots & a_0b_{2n-1} & a_0b_{2n} \\
 & a_1b_0 & \cdots & a_1b_{n-2} & a_1b_{n-1} & a_1b_n & a_1b_{n+1} & \cdots & a_1b_{2n-2} & a_1b_{2n-1} \\
 & & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
 & & & a_{n-1}b_0 & a_{n-1}b_1 & a_{n-1}b_2 & a_{n-1}b_3 & \cdots & a_{n-1}b_n & a_{n-1}b_{n+1} \\
 & & & & a_nb_0 & a_nb_1 & a_nb_2 & \cdots & a_nb_{n-1} & a_nb_n \\
\hline
 & & & & & a_{n+1}b_0 & a_{n+1}b_1 & \cdots & a_{n+1}b_{n-2} & a_{n+1}b_{n-1} \\
 & & & & & & a_{n+2}b_0 & \cdots & a_{n+2}b_{n-3} & a_{n+2}b_{n-2} \\
 & & & & & & & \ddots & \vdots & \vdots \\
 & & & & & & & & a_{2n-1}b_0 & a_{2n-1}b_1 \\
 & & & & & & & & & a_{2n}b_0
\end{array}
$$

Figure 2: Multiplication — before normalizing

$$
\begin{array}{cccc|ccccc|c}
c_{00} & c_{01} & \cdots & c_{0,n-1} & c_{0n} & c_{0,n+1} & c_{0,n+2} & \cdots & c_{0,2n-1} & c_{0,2n} \\
\hline
 & & & & & a_{n+1}b_0 & a_{n+1}b_1 & \cdots & a_{n+1}b_{n-2} & a_{n+1}b_{n-1} \\
 & & & & & & a_{n+2}b_0 & \cdots & a_{n+2}b_{n-3} & a_{n+2}b_{n-2} \\
 & & & & & & & \ddots & \vdots & \vdots \\
 & & & & & & & & a_{2n-1}b_0 & a_{2n-1}b_1 \\
 & & & & & & & & & a_{2n}b_0
\end{array}
$$

Figure 3: Multiplication — after normalizing first $(n+1)$ lines

The choice of $n \in \mathbb{N}$ is discussed in Section 4.1.3 (Chapter 4). Given $n \in \mathbb{N}$, for all $m > n$ we write:

$$\sum_{i=0}^{m} a_i b_{m-i} = \sum_{i=0}^{n} a_i b_{m-i} + \sum_{i=n+1}^{m} a_i b_{m-i}$$

We have:

$$\begin{aligned}\sum_{m=0}^{\infty} c_m r^{-m} &= \sum_{m=0}^{\infty}\left(\sum_{i=0}^{m} a_i b_{m-i}\right) r^{-m}\\ &= \sum_{m=0}^{n}\left(\sum_{i=0}^{m} a_i b_{m-i}\right) r^{-m} + \sum_{m=n+1}^{\infty}\left(\sum_{i=0}^{n} a_i b_{m-i} + \sum_{i=n+1}^{m} a_i b_{m-i}\right) r^{-m}\\ &= \sum_{m=0}^{\infty}\left(\sum_{i=0}^{\min(m,n)} a_i b_{m-i}\right) r^{-m} + \sum_{m=n+1}^{\infty}\left(\sum_{i=n+1}^{m} a_i b_{m-i}\right) r^{-m} \qquad (21)\end{aligned}$$

Now the sums $\sum_{i=0}^{\min(m,n)} a_i b_{m-i}$ are bounded for all $m \in \mathbb{N}_0$

$$\left|\sum_{i=0}^{\min(m,n)} a_i b_{m-i}\right| \leq (n+1)\,\rho^2, \qquad (22)$$

so we can apply `reduce` to the sequence $\left(\sum_{i=0}^{\min(m,n)} a_i b_{m-i}\right)_{m\in\mathbb{N}_0}$. Having done so $m(n)$ times, where

$$m\,(n) = \left\lceil \log_r\left((n+1)\,\rho^2 - \rho + 1\right)\right\rceil - 1,$$

we shall obtain an equivalent sequence $(c_{0m})_{m\in\mathbb{N}_0}$ satisfying $|c_{0m}| \leq r + \rho - 1$ for all $m \in \mathbb{N}_0$, i.e.

$$\sum_{m=0}^{\infty}\left(\sum_{i=0}^{\min(m,n)} a_i b_{m-i}\right) r^{-m} = \sum_{m=0}^{\infty} c_{0m} r^{-m}, \text{ where } |c_{0m}| \leq r + \rho - 1. \qquad (23)$$

Returning to (21), we rewrite it in the form

$$\sum_{m=0}^{\infty} c_m r^{-m} = \sum_{m=0}^{n-1} c_{0m} r^{-m} + r^{-n} \sum_{m=0}^{\infty} c_m^{(1)} r^{-m},$$

where

$$\begin{aligned} c_0^{(1)} &= c_{0n} \\ c_m^{(1)} &= c_{0,n+m} + \sum_{i=n+1}^{n+m} a_i b_{n+m-i},\ m \in \mathbb{N} \end{aligned} \tag{24}$$

Proceeding recursively with the series

$$\sum_{m=0}^{\infty} c_m^{(j)} r^{-m} = \sum_{m=0}^{n-1} c_{jm} r^{-m} + r^{-n} \sum_{m=0}^{\infty} c_m^{(j+1)} r^{-m},\ j \in \mathbb{N},$$

we obtain an equivalent sequence $(c'_m)_{m \in \mathbb{N}_0}$:

$$\begin{aligned} \sum_{m=0}^{\infty} c_m r^{-m} &= \sum_{m=0}^{n-1} c_{0m} r^{-m} + r^{-n} \left(\sum_{m=0}^{n-1} c_{1m} r^{-m} + r^{-n} \left(\sum_{m=0}^{n-1} c_{2m} r^{-m} + \dots \right.\right. \\ &= \sum_{m=0}^{n-1} c_{0m} r^{-m} + \sum_{m=n}^{2n-1} c_{1,m-n} r^{-m} + \sum_{m=2n}^{3n-1} c_{2,m-2n} r^{-m} + \dots \\ &= \sum_{k=0}^{\infty} \left(\sum_{m=kn}^{(k+1)n-1} c_{k,m-kn} r^{-m} \right) = \sum_{m=0}^{\infty} c'_m r^{-m}, \end{aligned}$$

where

$$\begin{aligned} c'_m &= c_{m \operatorname{div} n, m \operatorname{mod} n},\ |c'_m| \le r + \rho - 1,\ m \in \mathbb{N}_0 \\ (c_{km})_{m \in \mathbb{N}_0} &\sim \left(\left(\sum_{i=0}^{\min(m,n)} d_{i,m-i}^{(k)} \right)_{m \in \mathbb{N}_0} \right),\ |c_{km}| \le r + \rho - 1 \\ d_{ij}^{(k+1)} &= \begin{cases} c_{k,j+n}, & i = 0 \\ d_{i+n,j}^{(k)}, & i \in \mathbb{N} \end{cases} \\ d_{ij}^{(0)} &= a_i b_j \\ c_m^{(k+1)} &= \sum_{i=0}^{m} d_{i,m-i}^{(k+1)} \\ c_m^{(0)} &= c_m \end{aligned}$$

Finally, $(c'_m)_{m\in\mathbb{N}_0}$ can be normalized in a single application of `reduce`, which is the required result of multiplication[3].

2.4.3 Division

The intention in this chapter is to develop algorithms for division of exact real numbers. Division is the last, most complex, and most time-consuming of the four basic arithmetic operations. There is an infinite number of ways to perform division, and a plethora of division algorithms can be found in the literature [57, 68, 74, 77, 82]. As division may be considered the mathematical inverse of multiplication, we may reasonably expect to find algorithms that closely correspond to the inverse of the algorithms for multiplication. In order to synthesize the essential steps required for a division algorithm, we shall proceed by first examining ordinary paper-and-pencil division, and then showing how it could be adapted to the infinite precision case.

For the moment, let us assume that the dividend (numerator) N and the divisor (denominator) D are positive integral numbers. Even in this simplest case, division, unlike the other three operations of arithmetic, has a result consisting of two components: a quotient Q and a remainder R, such that $0 \leq R < D$ and $N = Q \cdot D + R$. The central idea of the paper-and-pencil method is easily stated: at each step, a digit of the quotient is determined, and a corresponding multiple of the divisor is subtracted from the partial remainder, which is initially equal to the dividend, and at the completion of the division — to the remainder. For

[3]Note that we have singled out the last step because of the possible need to adjust the exponent in case c_0 is not within the bounds

example, to divide 139/6 we perform

$$\begin{array}{rr|ccccl}
 & & 2 & 3 & & & = Q \\ \cline{3-6}
D = 6 & & 1 & 3 & 9 & & = N \\
 & - & 1 & 2 & & & \\ \cline{3-4}
 & & & 1 & 9 & & \\
 & & - & 1 & 8 & & \\ \cline{4-5}
 & & & & 1 & & = R
\end{array}$$

The presentation of the algorithm is no more difficult when the dividend and divisor, as well as the quotient and the remainder, are interpreted as (finite) fractions. Multiplication and division differ from addition and subtraction in that it is not necessary to first align the radix points: we can always reduce the original operands to integral ones, multiply or divide, and then determine the position of the radix point in the result by adding or subtracting the powers of the scale factors. Therefore, in order to obtain the fractional quotient Q, we similarly perform the division as a sequence of multiply-and-subtract cycles.

In hardware structures, the relatively expensive multiplications of the divisor by the digits of the quotient are usually done by repeated subtraction:

$$\left.\begin{array}{r}
13 \\
-6 \\ \hline
7 \\
-6 \\ \hline
1
\end{array}\right\} \text{Two subtractions of the divisor before overdrawing}$$

$$\left.\begin{array}{r}
-6 \\ \hline
-5
\end{array}\right\} \text{Overdraft}$$

After performing three subtractions of the divisor, the last one leaving a negative result, we can see that the first quotient digit is 2. At this point, we can either "restore" the partial remainder by adding the value that has just been subtracted (*the restoring method of division*), or avoid the restoration and leave it as it is (*non-restoring division*) by agreeing to *add* the divisor whenever the partial remainder is negative and *subtract* whenever it is positive. In the non-restoring method, we count the number of subtractions as a positive digit of the quotient, and the number of additions as a negative digit of the quotient. Thus non-restoring division may be viewed as being based on a signed-digit set $\{\overline{\rho}, \ldots, \overline{1}, 0, 1, \ldots, \rho\}$. The procedure shown in Fig. 4 consists of a sequence of subtractions/additions and shifts and is similar to that for multiplication. The quotient Q which has

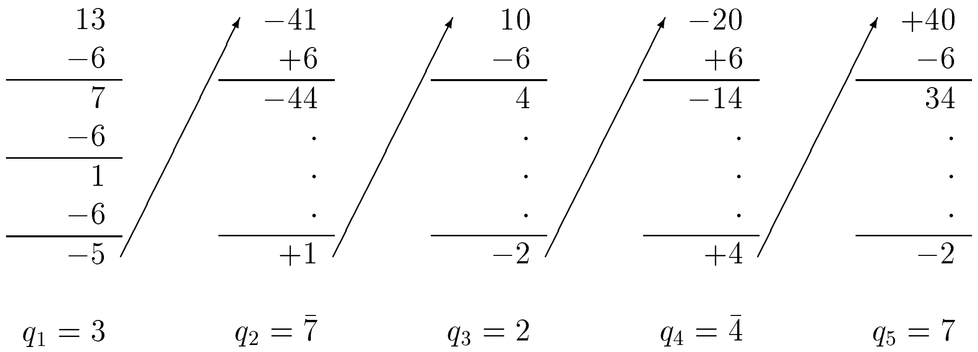


Figure 4: Non-restoring division

both positive and negative digits is given by

$$Q = 3\overline{7}.2\overline{4}7\overline{4}7 \cdots = 23.16666 \cdots$$

In the infinite precision case, the digits of the dividend and the divisor are usually unknown in advance but are available serially, in an online fashion: digit-by-digit, most significant digit first [82]. As with other arithmetic operations, the use of a redundant representation of the operands is required in order for the result to be computable and to bound the delay between input and output.

Let $N \in \mathbb{R}$ be the dividend, $D \in \mathbb{R}$, $D \neq 0$ — the divisor, their redundant signed-digit radix-r representations given by

$$N = r^{E_N} \cdot \sum_{i=0}^{\infty} n_i r^{-i}, \quad D = r^{E_D} \cdot \sum_{i=0}^{\infty} d_i r^{-i},$$

where $|n_i| \leq \rho$, $|d_i| \leq \rho$ for $i \in \mathbb{N}$. The task is to compute a real quotient

$$Q = r^{E_Q} \cdot \sum_{i=0}^{\infty} q_i r^{-i}$$

such that $N = Q \cdot D$ and $|q_i| \leq \rho$, $i \in \mathbb{N}$.

The majority of signed-digit division algorithms described in the literature in one way or another owe their origin to an algorithm due to Robertson [68]. The substance of the algorithm lies with an iterative process that produces one digit of the quotient per cycle according to the following recurrence equation, which is similar to the paper-and-pencil method described above:

$$P_{n+1} = r\ (P_n - q_n D)\,,\ n \in \mathbb{N}_0, \tag{25}$$

where $P_0 = N$, P_n is the current partial remainder, P_{n+1} is the next partial remainder, and q_n is the quotient digit inferred from P_n and D. It is easy to see that

$$P_n = r^n \left[N - \left(q_0 + q_1 r^{-1} + \ldots + q_{n-1} r^{-n+1}\right) D\right],\ n \in \mathbb{N},$$

and so imposing an upper bound on the value of $|P_n|$ will ensure convergence of the algorithm, provided that selection of the quotient digit q_n results in the next partial remainder P_{n+1} adhering to the same allowed range as P_n.

The existing signed-digit division algorithms primarily differ in their selection of quotient digits, restriction of the range of the possible values of the divisor, dividend and partial remainders and, finally, normalization techniques.

The conventional, non-redundant algorithms also use relation (25) but always produce *correct* quotient digits depending on the sign of a partial result as described above. Since the multiplications of the divisor by the digits of the quotient are done by repeated subtraction, a guessed digit is known to be incorrect if it is either too large and the subsequent subtraction leaves a negative result, or it is too small and the subtraction leaves a result that exceeds a multiple of the divisor in that digit position.

In redundant signed-digit representations, however, the sign of an intermediate result may not be readily available for inspection, because a number of its most significant digits, generally unknown in advance, may happen to be all zero. The usual way to get around this problem is to make a *guess* about q_n based on the inspection of several most significant digits of P_n and D. Even though this could result in some quotient digits q_n selected in this way being incorrect, the redundancy allows recovery from wrong guesses by taking an appropriate correction step in the next quotient digit. As long as the next q can correct an error in the previous step, convergence of the algorithm is guaranteed.

The method of division put forward here is a modification of the original Robertson's signed-digit division algorithm and is similar to that reported by David Smith [77].

The algorithm uses the above recurrence relation (25) and the following digit selection function:

$$q_n = \left\lfloor \left| \frac{p_{n0}}{d_0} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{p_{n0}}{d_0}\right), \tag{26}$$

where p_{n0} is the first digit of the n-th partial remainder P_n, and d_0 is the first digit of the divisor D which, being non-zero[4], is so scaled that $|d_0| \geq r^2$. Beginning with $P_0 = N$, we have the following sequences of digits representing P_{n+1}, $n \in \mathbb{N}_0$:

$$\begin{aligned} P_{n+1} &= r \cdot (p_{n0} - q_n d_0,\, p_{n1} - q_n d_1,\, \cdots,\, p_{nk} - q_n d_k,\, \cdots)_r \\ &= (r(p_{n0} - q_n d_0) + (p_{n1} - q_n d_1),\, p_{n2} - q_n d_2,\, \cdots,\, p_{nk} - q_n d_k,\, \cdots)_r \end{aligned}$$

The time spent normalizing partial results could be greater than the time spent generating them, so minimizing normalization is important. The early algorithms

[4]Note that since D is represented by an infinite sequence of digits, one cannot effectively check whether or not it is non-zero.

fully normalized P_n, $n \in \mathbb{N}$ at each step in order to keep the entries of the sequence bounded. However, as recently shown in [77] (and also suggested by Carl Pixley in the early 1980's), it is possible to skip the full normalization of the partial remainders, and instead normalize only a few leading digits. The details of the algorithm analysis are given in [77], and although considering the operands to be finite and given in non-redundant form, readily lend themselves to the elaboration necessary to extend the method to operate with infinite sequences of signed digits.

The elimination of most intermediate digit normalizations makes the division algorithm run in double-quick time, and at high precision even faster than multiplication (see Section 4.2.3).

Division by 2 when r is even

When the radix r is an even number, there exists a simple recursive procedure for dividing a number by 2. The idea should become clear if we note that for any $d \in \mathbb{Z}$,

$$\frac{d}{2} = \begin{cases} \left\lfloor \frac{d}{2} \right\rfloor, & \text{if} \quad d \equiv 0 \bmod 2 \\ \left\lfloor \frac{d}{2} \right\rfloor + \frac{1}{2}, & \text{if} \quad d \equiv 1 \bmod 2 \end{cases} \tag{27}$$

Since $r \equiv 0 \bmod 2$, we can write

$$\left\lfloor \frac{d}{2} \right\rfloor + \frac{1}{2} = \left\lfloor \frac{d}{2} \right\rfloor + \frac{r}{2} \cdot r^{-1},$$

where both numbers $\lfloor d/2 \rfloor$ and $r/2$ are integers. Introducing the notation

$$d' = \left\lfloor \frac{d}{2} \right\rfloor, \quad d'' = \begin{cases} 0, & \text{if} \quad d \equiv 0 \bmod 2 \\ r/2, & \text{if} \quad d \equiv 1 \bmod 2 \end{cases}, \tag{28}$$

we can rewrite (27) in the following form:

$$\frac{d}{2} = d' + d'' \cdot r^{-1}. \tag{29}$$

Now let $a \in \mathbb{R}$ be the dividend with the following redundant signed-digit radix-r representation:

$$a = r^{Ea} \cdot \sum_{i=0}^{\infty} a_i r^{-i},$$

where $|a_i| \leq \rho$ for $i \in \mathbb{N}$. It is obvious that

$$\frac{a}{2} = r^{Ea} \cdot \sum_{i=0}^{\infty} \frac{a_i}{2} r^{-i},$$

and using (29), we have

$$\frac{a}{2} = r^{Ea} \cdot (a'_0 + (a''_0 + a'_1)r^{-1} + (a''_1 + a'_2)r^{-2} + \dots, \tag{30}$$

where the sequences $(a'_k)_{k\in\mathbb{N}_0}$ and $(a''_k)_{k\in\mathbb{N}_0}$ are calculated according to (28). As with normalization, the computation of these sequences can be performed totally in parallel (cmp. Figure 1 and equations (13) and (15)). The final sequence (30) must undergo one pass of normalization, since $\left|a''_n + a'_{n+1}\right|$ may in general be greater than ρ. Note that by Theorem 3, one step of normalization suffices, because $\left|a''_n + a'_{n+1}\right| < r + \rho - 1$.

Example 3. *Let $r = 10$, $\rho = 6$. Compute $\sqrt{11}/2$.*

Using the method of square root function evaluation that will be described in Section 2.5.2, we obtain the following redundant radix-10 expansion of $\sqrt{11}$:

$$\sqrt{11} = (0, [3, 3, 2, \overline{3}, \overline{4}, 2, 5, \overline{2}, \overline{1}, 0, 3, 5, 5, 4, 0, 0, \overline{1}, \overline{5}, \overline{1}, 1, 2, \dots]).$$

According to (28),

$$(a'_k)_{k\in\mathbb{N}_0} = [1, 1, 1, \overline{2}, \overline{2}, 1, 2, \overline{1}, \overline{1}, 0, 1, 2, 2, 2, 0, 0, \overline{1}, \overline{3}, \overline{1}, 0, 1, \dots]$$
$$(a''_k)_{k\in\mathbb{N}_0} = [5, 5, 0, 5, 0, 0, 5, 0, 5, 0, 5, 5, 5, 0, 0, 0, 5, 5, 5, 5, 0, \dots],$$

and using (30), we get

$$\sqrt{11}/2 = (0, [1, 6, 6, \overline{2}, 3, 1, 2, 4, \overline{1}, 5, 1, 7, 7, 7, 0, 0, \overline{1}, 2, 4, 5, \ldots],$$

which, after normalizing, gives us

$$\sqrt{11}/2 = [2, \overline{3}, \overline{4}, \overline{2}, 3, 1, 2, 4, \overline{1}, 5, 2, \overline{2}, \overline{2}, \overline{3}, 0, 0, \overline{1}, 2, 4, 5, \ldots].$$

This method is also extensible to divisors that are prime factors of r other than 2, although for practical purposes it is often easier to use the general division method.

2.5 Evaluation of elementary functions

In this section, we shall discuss the evaluation of elementary functions on exact real numbers. Functions of real variables that can be defined for normalized signed-digit radix-r representations are precisely those for which there exist left-to-right algorithms defined on representations. These algorithms must work in an online fashion: digit-by-digit, most significant digit first, inputting digits of the argument(s) and outputting digits of the result with bounded delay. The question one should ask himself when defining a function on representations is whether, given more digits of the argument, one can produce more digits of the result. In particular, only continuous functions on exact reals are computable [66].

2.5.1 Some simple functions

Absolute value

The absolute value is probably one of the simplest functions definable on the real numbers. In floating-point systems, all that is required for its computation is the reversal of a number's sign bit, if need be — an operation so trivial that it is never even considered as such.

In exact real arithmetic systems, however, there is no algorithm for deciding whether or not two infinite sequences represent the same number. In particular, the predicates $=$, $<$ and $>$ are non-computable, and in general one cannot even check a number to see whether it is positive, negative, or zero.

Nonetheless, the absolute value function is *definable* on exact reals. Let us show that if the signed-digit radix-r system used is *not* maximally redundant, i.e. $\rho < r-1$, the sign of a number is determined by the sign of the first non-zero entry of its mantissa. Indeed, if a_k is the first non-zero element of $(a_n)_{n\in\mathbb{N}_0}$, then

$$r^{-k}\left(a_k - \frac{\rho}{r-1}\right) \le \sum_{n=0}^{\infty} a_n r^{-n} \le r^{-k}\left(a_k + \frac{\rho}{r-1}\right),$$

and if the system is not maximally redundant, all of these numbers have the same sign as a_k (provided $a_k \neq 0$). From this also results the conclusion that in non-maximally-redundant systems zero is represented *uniquely* (up to differences in exponents).

The algorithm for evaluation of the absolute value is now obvious:

$$\operatorname{abs}(a_0,\, a_1,\, \cdots,\, a_n, \cdots) = \begin{cases} 0 : \operatorname{abs}(a_1,\, a_2,\, \cdots,\, a_n, \cdots), & \text{if } a_0 = 0 \\ (a_0,\, a_1,\, \cdots,\, a_n, \cdots), & \text{if } a_0 > 0 \\ (-a_0,\, -a_1,\, \cdots,\, -a_n, \cdots), & \text{otherwise} \end{cases}$$

and its complexity is that of negation.

Round and non-deterministic conditionals

A major problem with *infinite* redundant radix-r representations is that there is no algorithm for deciding whether two number representations represent the same number. Since many programming tasks clearly require computable equality tests, it is obvious that any implementation of exact real arithmetic must have a datatype in which numbers are represented finitely. We shall discuss the choices for the set of finitely represented numbers in the design and implementation section of Chapter 4.

In this section, we propose the function, `round` n x, which approximates any representation with a finite one that differs from the argument by at most r^{-n}, where n is usually a positive integer. The function is defined on normalized sequences by

$$\texttt{round}\ n\ (E, (a_k)_{k\in\mathbb{N}_0}) = (E, (a_0, \ldots, a_n)),$$

where $(a_0, \ldots, a_n)$ is a shorthand for $(a_0, \ldots, a_n, 0, \ldots, 0, \ldots)$. This definition is, unfortunately, representation-dependent — if there are two normalized representations $(E, (a_k)_{k\in\mathbb{N}_0})$ and $(E', (a'_k)_{k\in\mathbb{N}_0})$ representing the same number, the value of `round` $n\ (E, (a_k)_{k\in\mathbb{N}_0})$ is not necessarily equal to the value of `round` $n\ (E', (a'_k)_{k\in\mathbb{N}_0})$.

The `round` function can also be used to define various useful non-deterministic comparison operators, such as

$$\texttt{LessOrEqual}\ n\ x\ y = ((\texttt{round}\ n\ x) \leq y),$$
$$\texttt{GreaterOrEqual}\ n\ x\ y = ((\texttt{round}\ n\ x) \geq y),$$

where y is a *finite* number, so that

$$\texttt{LessOrEqual}\ n\ x\ y = \begin{cases} \texttt{True}, & \text{if} \quad x \leq y - r^{-n} \\ \texttt{False}, & \text{if} \quad x > y + r^{-n} \\ \texttt{True/False}, & \text{if} \quad y - r^{-n} < x \leq y + r^{-n} \end{cases}$$

When $y - r^{-n} < x \leq y + r^{-n}$, the given precision is insufficient to reliably determine whether or not $x \leq y$, and the comparison function can return either `True` or `False` (hence non-determinism).

Minimum and maximum

It may come as a surprise to some to learn that while the comparison operators $<$ and $>$ are clearly non-computable on exact reals, the functions minimum and maximum are. This is most readily seen from the relations

$$\min(a,b) = \frac{a+b-|a-b|}{2},$$
$$\max(a,b) = \frac{a+b+|a-b|}{2},$$

which involve only computable functions: addition, subtraction, absolute value, and division by 2.

The implications of computability of min and max are non-trivial: for example, we can sort lists of exact real numbers using sorting algorithms based upon max and min, rather than upon $<$ and $>$ (such as Batcher's merge sort).

Polynomials and rational functions

Polynomial and rational approximations have always played a prominent rôle in the study of function evaluation routines. In floating-point arithmetic, the methods for evaluation of special functions are usually connected in one way or another with the study of such approximations. Needless to say, function approximation techniques are inherently inaccurate, and in the following sections we shall describe methods of function evaluation that produce *exact* results using *infinite series* instead of finite approximations.

Nevertheless, it is not uncommon in computational practice to use polynomials and rational functions for purposes other than function approximation, and therefore we include them in our library of functions. There are two kinds of manipulations that one can do with a polynomial — numerical manipulations (e.g. evaluation at some argument), or algebraic manipulations (e.g. rearranging the coefficients in some way without choosing any particular argument). Algebraic manipulations are beyond the scope of this thesis.

Polynomials

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0 \tag{31}$$

are never evaluated directly (by computing the n-th power of x, multiplying by a_n, etc). The most frequently used polynomial evaluation method is the technique called *Horner's scheme* or *nested multiplication*, according to which an n-th degree polynomial (31) is represented in the form

$$P_n(x) = (\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0,$$

and the computation proceeds in the order indicated by the brackets. Nested multiplication requires $nM + nA$ operations (n multiplications and n additions) for the evaluation of an arbitrary n-th degree polynomial, and is widely used by numerical analysts not just because of its simplicity, but also because of its numerical reliability [31]. Many other polynomial evaluation methods exist [29, 49] by means of which an n-th degree polynomial can be evaluated with less than n multiplications, although such methods usually require slightly more than n additions.

One obvious disadvantage of this method is that it starts at the coefficient of the highest power of x, which makes it unsuitable for the evaluation of infinite power series, which normally proceeds in the opposite direction.

Rational functions

$$R(x) = \frac{P_n(x)}{Q_m(x)} = \frac{p_n x^n + p_{n-1}x^{n-1} + \ldots + p_1 x + p_0}{q_m x^m + q_{m-1}x^{m-1} + \ldots + q_1 x + q_0}$$

are evaluated in the obvious way — as two polynomials, followed by a divide.

Integer power function

The power function x^n, $n \in \mathbb{N}$, is a polynomial of a special form. The computation of the power x^n by Horner's rule involves $n-1$ multiplications $x \cdot x \cdot \ldots \cdot x$. It is obvious, however, that in order to compute, say x^8, we could form successive factors

$$x^2 = x \cdot x, \quad x^4 = x^2 \cdot x^2, \quad x^8 = x^4 \cdot x^4,$$

which requires only three multiplications instead of seven.

A fast method for the evaluation of powers is the binary exponentiation method, which evaluates x^n using the binary representation of $n \in \mathbb{N}$ by either squaring, or multiplying the current result by x, depending on the current bit (1 or 0) in the binary expansion of n. For example, to evaluate x^{12} we note that $n = 12_{10} = 1100_2$ and successively compute $x^2 = x \cdot x$, $x^3 = x^2 \cdot x$, $x^6 = x^3 \cdot x^3$, $x^{12} = x^6 \cdot x^6$. The number of multiplications required is therefore less than or equal to $2(N_b - 1)$, where N_b is the number of bits in the binary representation of n.

The binary method does not in fact give the minimum possible number of multiplications — the smallest counterexample is $n = 15$, when the binary method needs 6 multiplications ($x^2 = x \cdot x$, $x^3 = x^2 \cdot x$, $x^6 = x^3 \cdot x^3$, $x^7 = x^6 \cdot x$, $x^{14} = x^7 \cdot x^7$, and $x^{15} = x^{14} \cdot x$), while the same result can be achieved with only five multiplications: $x^2 = x \cdot x$, $x^3 = x^2 \cdot x$, $x^6 = x^3 \cdot x^3$, $x^{12} = x^6 \cdot x^6$, $x^{15} = x^{12} \cdot x^3$. More information on exponentiation algorithms can be found e.g. in [45].

2.5.2 Square rooting

The square root function is singled out because of its simplicity and amenability to implementation with little additional overhead beyond that of the basic arithmetic operations. It is also almost the only commonly used function that is evaluated iteratively. The algorithm that we will describe is the direct analogue of that for division and produces n digits of the result in n cycles, at a rate of one digit per cycle. Such pseudo-division methods can also be extensible to higher degrees, although roots of order greater than three are usually evaluated by the same methods as x^y for arbitrary y, using exponents and logarithms, and even cube-root functions are somewhat uncommon in function libraries. Our primary emphasis will therefore be on evaluation of $\sqrt{x}$.

Suppose that we want to evaluate $y = \sqrt{x}$ in radix r. Let x be given by a normalized signed-digit sequence $X = (x_0, x_1, \ldots, x_n, \ldots)_r$ with $x_0 > 0$, $|x_n| \leq \rho$,

$n \in \mathbb{N}$, and exponent e, so that $x = r^e \cdot \sum_{n=0}^{\infty} x_n r^{-n}$, $x > 0$. Then

$$y = \begin{cases} r^{e/2} \cdot \sqrt{X}, & \text{if } e \text{ is even} \\ r^{(e-1)/2} \cdot \sqrt{rX}, & \text{if } e \text{ is odd} \end{cases}$$

Let $Y = (y_0, y_1, \ldots, y_n, \ldots)$ be a mantissa of y such that $|y_n| \leq \rho$, $n \in \mathbb{N}$. Denote

$$Y_n = y_0 + y_1 r^{-1} + \ldots + y_n r^{-n}$$

so that $Y = \lim_{n\to\infty} Y_n$. Consider the scaled partial remainders

$$P_0 = X,\ P_n = r^n \left(X - Y_{n-1}^2\right),\ n \in \mathbb{N}.$$

Observing that $Y_n = Y_{n-1} + y_n r^{-n}$, we can rewrite the next partial remainder as

$$P_{n+1} = r^{n+1}\left(X - Y_n^2\right) = r\left(P_n - 2y_n Y_{n-1} - y_n^2 r^{-n}\right)$$

The square root algorithm is based on the above recurrence relation, each iteration of which consists of two subcomputations:

1) Determination of the result digit y_n using a digit selection function s, which has P_n and Y_{n-1} as arguments:

$$y_n = s(P_n, Y_{n-1})$$

2) Formation of P_{n+1} from P_n, Y_{n-1} and y_n:

$$P_{n+1} = r\left(P_n - 2y_n Y_{n-1} - y_n^2 r^{-n}\right) \tag{32}$$

If care is exercised in choosing y_0 and the selection function s is such that

$$\ldots \leq |P_{n+1}| \leq |P_n| \leq \ldots \leq |P_1| \leq |X|,$$

the algorithm will converge as

$$\left|X - Y_n^2\right| = \frac{1}{r^{n+1}}\left|P_{n+1}\right| \le \frac{1}{r^{n+1}}\left|X\right|.$$

As in the case of division, we make guesses about the digits y_n based on the most significant digits of the current remainder P_n and the current approximation Y_{n-1}. Although the guessed digits may be incorrect in some cases, no correction steps would be needed if a redundant signed-digit representation of P_n was used. In particular, it can be shown that the following digit selection function

$$y_n = \left\lfloor \frac{|p_{n0}|}{2y_0} \right\rfloor \cdot \operatorname{sgn} p_{n0}, \tag{33}$$

where p_{n0} is the integer part of P_n and $y_0 = \lfloor \sqrt{x_0} \rfloor$ the first digit of Y, is reliable enough as long as $y_0 \ge \lfloor \sqrt{r} \rfloor$ (i.e. $x_0 \ge r$), and the exponent of x is even.

Example 4. *Compute $\sqrt{2}$ in radix $r = 10$, $\rho = 6$.*
First, we convert the argument, $2 = (0, [2])$, to a special normalized form with an even exponent and $x_0 \ge r$:

$$P_0 = (-2, [200]),$$

and then apply the square rooting procedure to its mantissa, $[200]$ (we already know that the exponent of the result will be -1 and henceforth can deal with mantissas only). The first digit y_0 is computed as a rounded down floating-point approximation of $\sqrt{200}$ (we assume the existence of an inexpensive low-precision hardware floating-point square root operation):

$$y_0 = \left\lfloor \sqrt{200} \right\rfloor = 14.$$

Using (32), we compute the next partial remainder

$$P_1 = 10 \cdot (P_0 - y_0^2) = 10 \cdot ([200] - [196]) = 10 \cdot [4] = [40],$$

and guess the next digit according to (33):

$$y_1 = \left\lfloor \frac{40}{2 \cdot 14} \right\rfloor = 1.$$

The subsequent remainders become slightly more complicated due to the presence of a non-zero Y_{n-1} in (32), thus,

$$\begin{aligned} P_2 &= 10 \cdot (P_1 - 2y_1y_0 - y_1^2 10^{-1}) = 10 \cdot ([40] - [28] - [0,1]) = \\ &= 10 \cdot [12, -1] = [119], \end{aligned}$$

which gives us the next digit

$$y_2 = \left\lfloor \frac{119}{2 \cdot 14} \right\rfloor = 4.$$

Similarly, we have

$$\begin{aligned} P_3 &= 10 \cdot (P_2 - 2y_2(y_0 + y_1 10^{-1}) - y_2^2 10^{-2}) = \\ &= 10 \cdot ([119] - 2 \cdot [4] \cdot ([14] + [0,1]) - [0,0,16]) = \\ &= 10 \cdot ([119] - [112, 8] - [0,0,16]) = \\ &= 10 \cdot ([7, -8, -16]) = [62, -16] = [61, -6], \end{aligned}$$

and

$$y_3 = \left\lfloor \frac{61}{2 \cdot 14} \right\rfloor = 2. \qquad (34)$$

Continuing in this manner ad infinitum, we find

$$\sqrt{2} = (-1, [14, 1, 4, 2, 1, 3, 6, \overline{4}, 2, 4, \overline{3}, \ldots]) = (0, [1, 4, 1, 4, 2, 1, 3, 6, \overline{4}, 2, 4, \overline{3}, \ldots]).$$

2.5.3 Limits

In this section it will be our aim to show how to compute the sum of a convergent infinite series $\sum_{n=0}^{\infty} a_n$ of computable radix-r numbers. We find the properties that sequence $(a_n)_{n\in\mathbb{N}_0}$ must possess in order for the sum to be computable, and show that the class of such sequences is wide enough to enable us to compute a number of important elementary functions. The method we put forward also lends itself to computation of limits of sequences, and we clarify the differences between them.

Summation of infinite series

We shall consider series $\sum\limits_{k=0}^{\infty} a_k$, which is known to be absolutely convergent, and whose terms $a_k \neq 0$ are monotonic decreasing (in the weak sense):

$$|a_{n+1}| \leq |a_n|, \quad n \in \mathbb{N}_0.$$

Assume that a_n are normalized radix-r numbers

$$a_n = (e_n, [a_{n0}, a_{n1}, \cdots, a_{nk}, \cdots]), \ |a_{nk}| \leq \rho; \quad n, k \in \mathbb{N}_0, \tag{35}$$

and $a_{k0} \neq 0$ for all $k \in \mathbb{N}_0$ (this is always possible if $a_k \neq 0$). Then the exponents e_n must also be non-increasing:

$$e_0 \geq e_1 \geq \ldots \geq e_n \geq \ldots, \tag{36}$$

and $e_n \underset{n\to\infty}{\to} -\infty$ (since $\lim\limits_{n\to\infty} a_n = 0$ for any convergent series $\sum\limits_{k=0}^{\infty} a_k$). Note that we can assume $e_0 = 0$ without any loss of generality.

Let us introduce the following notation:

$$\begin{aligned} n_0 &= \max\{k \in \mathbb{N} \mid e_{k-1} = e_0\} \\ n_1 &= \max\{k \in \mathbb{N} \mid e_{n_0+k-1} = e_{n_0}\} \\ n_2 &= \max\{k \in \mathbb{N} \mid e_{n_0+n_1+k-1} = e_{n_0+n_1}\} \\ &\cdots \end{aligned} \tag{37}$$

Due to the earlier property (36), the maxima in (37) all exist, and therefore the numbers n_k are correctly defined.

The sequence $(n_k)_{k\in\mathbb{N}_0}$ is quite important, as it enables us very simply to obtain a great deal of information about the rate of convergence (and therefore, online computability) of the original series. Indeed, as is clear from the definition, n_k are but the number of times each exponent e_k appears in the sequence $(e_k)_{k\in\mathbb{N}_0}$, so that (36) can be rewritten as follows

$$\underbrace{e_0\ e_0\ \cdots\ e_0}_{n_0}\ \underbrace{e_{n_0} e_{n_0} \cdots\ e_{n_0}}_{n_1}\ \underbrace{e_{n_1} e_{n_1} \cdots\ e_{n_1}}_{n_2}\ \cdots$$

It is clear that in order to compute $\sum_{k=0}^{\infty} a_n$ online with a bounded delay, the numbers n_k should not be allowed to grow uncontrollably. Indeed, we have

$$|a_n| \leq r^{e_n} \frac{\rho r}{r-1}, \quad n \in \mathbb{N}_0,$$

so

$$\begin{aligned}
\sum_{n=0}^{\infty} a_n \leq \sum_{n=0}^{\infty} |a_n| \leq \frac{\rho r}{r-1} &\left(\underbrace{r^{e_0} + \cdots + r^{e_0}}_{n_0} + \underbrace{r^{e_{n_0}} + \cdots + r^{e_{n_0}}}_{n_1} + \ldots \right) \leq \\
&\leq r^{e_0} \frac{\rho r}{r-1} \left(n_0 + n_1 r^{e_{n_0} - e_0} + n_2 r^{e_{n_1} - e_0} + \ldots \right) \leq \qquad (38) \\
&\leq r^{e_0} \frac{\rho r}{r-1} \sum_{k=0}^{\infty} n_k r^{-k}, \qquad (39)
\end{aligned}$$

provided the last series in (38) is convergent.

A naïve algorithm would start with e_0 and scan the sequence (35) until it finds $e_{n_0} < e_0$, then add up $\sum_{k=0}^{n_0-1} a_k$, print $(e_0 - e_{n_0})$ digits of the sum, shift the remainder by $(e_0 - e_{n_0})$ positions, scan the sequence further until it finds $e_{n_1} < e_{n_0}$, add the remainder to $\sum_{k=n_0}^{n_1-1} a_k$, print the next $(e_{n_0} - e_{n_1})$ digits, and proceed recursively. Formally speaking, we reduce (35) to an equivalent sequence

$$b_k = \left(e'_k, [b_{k0}, b_{k1}, \cdots, b_{kn}, \cdots]\right), \ |b_{kn}| \leq n_k \rho; \quad k, n \in \mathbb{N}_0, \qquad (40)$$

where

$$
\begin{aligned}
b_{0j} &= \sum_{i=0}^{n_0-1} a_{ij}, \quad e'_0 = e_0, \\
b_{1j} &= \sum_{i=n_0}^{n_0+n_1-1} a_{ij}, \quad e'_1 = e_{n_0}, \\
&\dots \\
b_{kj} &= \sum_{i=n_0+\ldots+n_{k-1}}^{n_0+\ldots+n_k-1} a_{ij}, \quad e'_k = e_{n_{k-1}},
\end{aligned}
$$

and $e'_0 > e'_1 > e'_2 > \ldots > e'_n > \ldots$

However, the resulting sequence of digits printed in this way would be unnormalized, and its normalization requires knowledge of the behaviour of n_i, as $|b_{ij}| \leq n_i \rho$ for all $i, j \in \mathbb{N}_0$. We usually require that n_k be bounded for all $k \in \mathbb{N}_0$, which significantly simplifies the normalization process, albeit at the expense of restricting the class of series computable in this way. Indeed, according to (18), if $|n_k| \leq M$ for all $k \in \mathbb{N}_0$, the resulting sequence can be fully normalized in $n = \lceil \log_r(M\rho - \rho + 1) \rceil$ applications of `reduce`.

There is naturally nothing to stop our omitting this requirement and allowing n_k to grow, provided, of course, that we apply stronger normalization later. In practice, however, such series turn out to be very slow at converging, and whenever possible, an alternative series with bounded n_k should be sought. An example of such series is given below.

Example 5. *Consider the series*

$$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \ldots + \frac{1}{n(n+1)} + \ldots.$$

Here

$$a_n = \frac{1}{(n+1)(n+2)} = \frac{1}{n+1} - \frac{1}{n+2},$$

so that $A_n = \sum_{k=0}^{n} a_k = 1 - \frac{1}{n+2}$, *and it is obvious that* $\sum_{n=0}^{\infty} a_n = 1$, *as* $\lim_{n\to\infty} A_n = 1$.

Although the series does converge to 1, it converges extremely slowly, as can be seen from the following ($r = 10$, $\rho = 6$):

$$n_0 = 3,\ n_1 = 9,\ n_2 = 28,\ n_3 = 88,\ n_4 = 279,\ \ldots$$

The equivalent sequence (40) is obtained from a_k by adding $n_0 = 3$ terms with $e_0 = -1$, $n_1 = 9$ terms with $e_3 = -2$, $n_2 = 28$ terms with $e_9 = -3$, etc.

$$\begin{aligned}
b_0 = a_0 + \ldots + a_2 &= (-1, [8, -5, 0, 0, 0, 0, 0, \ldots])\\
b_1 = a_3 + \ldots + a_{11} &= (-2, [17, 2, 10, 7, 6, 7, 22, 2, 10, 7, \ldots])\\
b_2 = a_{12} + \ldots + a_{39} &= (-3, [53, -6, 11, 19, 38, 0, 30, 28, 21, \ldots])\\
b_3 = a_{40} + \ldots + a_{127} &= (-4, [166, -6, 86, 116, 64, 59, 63, 54, \ldots])\\
b_4 = a_{128} + \ldots + a_{406} &= (-5, [528, -7, 253, 237, 265, 196, 272, \ldots]\\
&\ldots
\end{aligned}$$

The resulting sequence produced by the algorithm is

$$a = (-1, [8, 12, 55, 170, 540, 1773, 5671, 17750, 56336, \ldots]),$$

which after 5 *applications of* `reduce` *becomes*

$$(0, [1, 0, 0, 0, -1, 7, \ldots],$$

producing only 4 *correct digits after the summation of thousands of terms* (-1 *appears in the sequence because the rest of it is still largely unnormalized).*

Fortunately, as we shall see later, most elementary functions have series expansions converging much faster than the series in this example.

So far we have said little about how to decide whether a given series is convergent or divergent, and we have had to apply *ad hoc* methods to our examples. The short answer to this question is that we cannot — just as it is impossible to check whether a divisor is non-zero, no algorithm can detect online that e.g. the harmonic series

$$1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \ldots$$

is divergent, as the convergence to zero of the sequence of terms does not suffice to ensure the convergence of the sequence of partial sums.

Sequences and their limits

Let $(a_n)_{n \in \mathbb{N}_0}$ be a sequence of computable radix-r numbers, converging to, say, $a \in \mathbb{R}$. Classically, this means that only the "very distant" members of the sequence tend to a, and in fact, we can discard any fixed number of sequence elements without affecting its limit. Naturally, if we would like to compute limits in a lazy fashion, we have to impose certain restrictions on the numbers in our sequence, lest we potentially have to wait for a very long time before we can compute even the first digit of the result.

Let us assume that

$$|a_{n+2} - a_{n+1}| \leq |a_{n+1} - a_n| \tag{41}$$

for all $n \in \mathbb{N}_0$. Denoting

$$s_0 = a_0, \quad s_n = a_n - a_{n-1},\ n \in \mathbb{N}, \tag{42}$$

we have

$$|s_{n+1}| \leq |s_n|, \quad n \in \mathbb{N},$$

and therefore, to evaluate $\lim_{n\to\infty} a_n$, we can apply the results of the previous section to the series

$$\sum_{n=0}^{\infty} s_n = \lim_{n\to\infty} a_n,$$

provided the limit exists.

The main difference between the computation of a sequence limit and that of a series is that we can start at any position $k \in \mathbb{N}$ in the sequence. To be specific, instead of starting with a_0 in (42), we could begin the computation with the element a_k by setting

$$s_0 = a_k, \quad s_n = a_n - a_{n-1},\ n > k.$$

In essence, the limit is a property of the "distant part" of a sequence: the beginning, or indeed any finite part of the sequence, is irrelevant. The converse statement is also true: no finite number of terms determines the limit, or existence of one, nor reveals anything relevant to the existence or value of the limit.

There is also no way for a computer to detect the situation when a sequence diverges, while condition (41) holds true. Consider, for instance, the following sequence

$$a_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}.$$

The series corresponding to this sequence is the harmonic series

$$\sum_{n=1}^{\infty} s_n = \sum_{n=1}^{\infty} 1/n,$$

which clearly diverges, yet $|s_{n+1}| \leq |s_n|$ for all $n \in \mathbb{N}$.

The programmer must therefore always guard against such fallacies and ensure that the argument to the limit function is a convergent sequence or series. It is also often possible to accelerate convergence of a sequence by discarding a fixed number of its entries and thus avoiding the summation of unnecessary terms.

2.5.4 Elementary transcendental functions

In this section we shall discuss evaluation methods for a number of elementary transcendental functions, such as the exponential, logarithmic, circular, hyperbolic, and some other related functions. The technique we shall use is to apply the results of the preceding section to power series expansions of these functions, combined with the use of range-reducing procedures. The need for range reduction is self-evident — although for many functions the full argument range is nominally an infinite interval such as $(0, \infty)$ or $(-\infty, \infty)$, a problem occurs with series that converge everywhere in the mathematical sense, but only in part of the full argument range fast enough to be practically manageable. Situations do arise in which it is possible to use a power series expansion satisfactorily over an infinite range; however, power series for functions like

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} \tag{43}$$

or

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} \tag{44}$$

do not even start to converge until $n \gg |x|$, which makes them useless for large ranges. Fortunately, as we will show, there are quite simple techniques for reducing the argument range over which such expansions can be used.

The exponential function

The exponential function a^x $(a > 0,\ a \neq 1)$ is defined everywhere on the real axis, it is continuous, convex and monotonically increasing for $a > 1$ (decreasing for $a < 1$). For any arbitrary base $a > 0$, the exponential function a^x may be transformed to the exponential function $\exp x = e^x$ (e is the basis of natural logarithm) by using the identity

$$a^x = e^{x \ln a}.$$

The computation of an exponential function may always be reduced to a computation for an argument in the range $(0, 1)$:

$$a^x = a^n \cdot a^y,$$

where $n = \lfloor x \rfloor$, $y = \{x\}$, $0 \leq y < 1$. Furthermore, we can reduce it to the range

$(0, 2^{-m})$ for any $m \in \mathbb{N}$:

$$a^y = (a^{y \cdot 2^{-m}})^{2^m} = \underbrace{((\cdots (a^{y \cdot 2^{-m}})^2)^2 \cdots)^2}_{m \text{ times}}.$$

The exponential function $\exp x$ has the following power series expansion

$$\exp x = \sum_{k=0}^{\infty} \frac{x^k}{k!}, \tag{45}$$

which has radius of convergence infinity. The coefficients $1/k!$ diminish very rapidly — faster than any power of x; however, before k is such that $x^k < k!$, the terms are increasing.

Let x be represented by a normalized signed-digit sequence $(x_0, x_1, \ldots, x_n, \ldots)_r$ with $|x_n| \le \rho$, $n \in \mathbb{N}_0$, and exponent e_x, so that $x = r^{e_x} \cdot \sum_{n=0}^{\infty} x_n r^{-n}$. If $e_x \le 0$, we can directly sum series (45); otherwise, we compute

$$\exp x = \left(\exp(\sum_{n=0}^{\infty} x_n r^{-n}) \right)^{r^{e_x}},$$

where r^{e_x} is an integer power and can be computed using the method discussed in Section 2.5.1.

Example 6. *Compute the constant $e = \exp 1$.*

We first compute the terms of power series (45) that form the sequence a_k (35):

$$\begin{array}{ll} a_0 = (0, [1]) & a_1 = (0, [1]) \\ a_2 = (-1, [5]) & a_3 = (-1, [2, \overline{3}, \overline{3}, \overline{3}, \ldots]) \\ a_4 = (-2, [4, 2, \overline{3}, \overline{3}, \overline{3}, \ldots]) & a_5 = (-2, [1, \overline{2}, 3, 3, 3, \ldots]) \\ a_6 = (-3, [1, 4, \overline{1}, \overline{1}, \overline{1}, \ldots]) & \\ a_7 = (-4, [2, 0, \overline{2}, 4, 1, 3, \overline{3}, 0, \overline{2}, \ldots]) & \\ a_8 = (-5, [2, 5, \overline{2}, 0, 1, 6, \overline{1}, \overline{3}, 3, 0, 1, \ldots]) & \end{array} \tag{46}$$

Each row in (46) holds the entries of a_k with the same exponent; therefore, the number of elements in each row is n_k (see (37)): $n_0 = 2$, $n_1 = 2$, $n_3 = 2$,

$n_4 = n_5 = n_6 = 1$, *etc. Adding the elements (one row at a time), we compute*

$$\begin{aligned}
b_0 &= (0, [2]) \\
b_1 &= (-1, [7, \overline{3}, \overline{3}, \overline{3}, \ldots]) = (0, [0, 7, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \ldots]) \\
b_2 &= (-2, [5, 0, 0, 0, \ldots]) = (0, [0, 0, 5, 0, 0, 0, 0, \ldots]) \\
b_3 &= (-3, [1, 4, \overline{1}, \overline{1}, \ldots]) = (0, [0, 0, 0, 1, 4, \overline{1}, \overline{1}, \ldots]) \\
b_4 &= (-4, [2, 0, \overline{2}, 4, \ldots]) = (0, [0, 0, 0, 0, 2, 0, \overline{2}, 4, \ldots]) \\
&\ldots
\end{aligned}$$

Finally adding up (and appropriately shifting) the b_k, *we obtain*

$$e = (0, [2, 7, 2, \overline{2}, 3, \overline{2}, 2, \overline{2}, 3, \overline{2}, \ldots]).$$

Note that since the n_k for this series are small (typically 1 or 2), no intermediate normalization of the b_k was necessary.

The logarithmic function

The logarithmic function $\log_a x$ (the logarithm of x to the base a), where $a > 0$ and $a \neq 1$, is the inverse of the exponential function:

$$a^{\log_a x} = x, \quad \log_a a^y = y.$$

The *natural logarithm* of x is its logarithm to the base e:

$$\ln x = \log_e x, \quad e^{\ln x} = x, \quad \ln e^y = y.$$

Similarly, the *decimal logarithm* of x is its logarithm to the base 10:

$$\log x = \log_{10} x.$$

The logarithmic function is defined everywhere in the interval $(0, +\infty)$, and is everywhere continuous in that interval. If $a > 1$, then $\log_a x$ is a concave monotonically increasing function ($\log_a x < 0$ if $x < 1$ and $\log_a x > 0$ if $x > 1$). If, on the other hand, $0 < a < 1$, then $\log_a x$ is a convex monotonically decreasing function, with $\log_a x > 0$ if $x < 1$ and $\log_a x < 0$ if $x > 1$.

There are simple connections between logarithms to different bases:

$$\begin{aligned} \log_a x &= \frac{1}{\log_x a}, \\ \log_a x &= -\log_{\frac{1}{a}} x, \end{aligned} \tag{47}$$

and in particular,

$$\log_a x = \frac{\ln x}{\ln a}. \tag{48}$$

The principal relations concerning logarithms are the following:

$$\begin{aligned} \log_a(xy) &= \log_a x + \log_a y, \\ \log_a x^\alpha &= \alpha \cdot \log_a x. \end{aligned}$$

Using these formulae, we can reduce the problem of evaluating a logarithm to base a for any positive number to that of finding the logarithm of a number in the range $(1, a)$ or the range $(1/a, 1)$. Indeed, for any number $x > 0$, there is a unique integer $n \geq 0$ such that

$$a^n \leq x < a^{n+1}.$$

Then

$$x = a^n y = a^{n+1} y', \quad 1 \leq y \leq a, \quad \frac{1}{a} \leq y' = \frac{1}{a} y < 1,$$

and

$$\log_a x = n + \log_a y = (n+1) + \log_a y'. \tag{49}$$

To summarize the above steps, in order to evaluate an arbitrary logarithm $\log_a x$ for some $x > 0$ and $a > 0$, $a \neq 1$, we first use (48) to reduce the problem to that of evaluating $\ln a$ and $\ln x$. The latter can be computed using the following power series expansion of the natural logarithm:

$$\ln x = 2\sum_{k=1}^{\infty} \frac{1}{2k-1}\left(\frac{x-1}{x+1}\right)^{2k-1}, \tag{50}$$

which holds for all $x > 0$.

Suppose that $x > 0$ is given by its canonical radix-r representation

$$x = r^{e_x} \cdot (x_0 + x_1 r^{-1} + \ldots + x_n r^{-n} + \ldots).$$

Naturally, we can assume that $x_0 > 0$ or else the logarithm of x is undefined. Then

$$x_0 + x_1 r^{-1} + \ldots + x_n r^{-n} + \ldots \leq \rho(1 + r^{-1} + \ldots + r^{-n} + \ldots) = \frac{\rho r}{r-1} < r,$$

and we have the following inequality

$$r^{e_x} \leq x < r^{e_x+1}.$$

Using (49), we can reduce the argument range to $(0, 1)$ observing that

$$\ln x = (e_x + 1)\ln r + \ln(x_0 r^{-1} + x_1 r^{-2} + \ldots + x_n r^{-n-1} + \ldots).$$

Example 7. *Compute* $\ln 2$ *in radix* 10.

According to (50),

$$\ln 2 = 2\sum_{k=1}^{\infty} \frac{1}{2k-1}\left(\frac{1}{3}\right)^{2k-1} = 2\left(\frac{1}{3} + \frac{1}{3}\left(\frac{1}{3}\right)^3 + \frac{1}{5}\left(\frac{1}{3}\right)^5 + \ldots\right),$$

and evaluating the terms of this series gives us

$$\begin{aligned} a_0 &= (0, [1, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \overline{3}, \ldots]) \\ a_1 &= (-2, [2, 5, \overline{3}, \overline{1}, 1, 4, \overline{4}, \overline{2}, 0, 2, 5, \overline{3}, \overline{1}, \ldots]) \\ a_2 &= (-3, [2, \overline{4}, 5, \overline{4}, 1, \overline{1}, 1, \overline{5}, 3, 5, 0, \overline{2}, \overline{1}, \ldots]) \\ &\ldots \end{aligned}$$

Adding up the a_n and normalizing, we obtain the value of

$$\ln 2 = (0, [1, \overline{3}, \overline{1}, 3, 1, 5, \overline{3}, 2, 2, \overline{2}, 1, \overline{4}, \overline{4}, 0, \overline{1}, \ldots]).$$

Computation of π

> Also, he made a molten sea of ten cubits from brim to brim, round in compass, and five cubits the height thereof; and a line of thirty cubits did compass it round about (I Kings 7:23)

The quest for π dates back to the beginning of recorded history, and in many ways parallels the progress of mathematics and science in general. As evidenced by the above passage from the Old Testament, the simple approximation of 3 was sufficient for the needs of the people of antiquity, and it was not until circa 250 B.C. that the improved bounds $3\frac{10}{71} < \pi < 3\frac{1}{7}$ were found by Archimedes who used geometrical methods based on inscribed and circumscribed polygons [35].

In the age of Newton and Leibniz, a number of substantially new formulae for π were discovered, given rise to by the discovery of calculus. The well-known Gregory-Leibniz formula

$$\frac{\pi}{4} = \arctan 1 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots,$$

was devised, which however could not be used to compute the value of π, as the series was so slow-converging that hundreds of terms would be required to compute π to even two digits' accuracy. A significantly faster formula was constructed by

Machin by utilizing the identity

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}.$$

Machin's formula was used by Shanks to compute π to 707 decimal digits' accuracy[5] in 1873. Sir Isaac Newton himself used the formula

$$\pi = \frac{3\sqrt{3}}{4} + 24\left(\frac{1}{3\cdot 2^3} - \frac{1}{5\cdot 2^5} - \frac{1}{7\cdot 2^7} - \frac{1}{9\cdot 2^9} - \dots\right)$$

to compute π to high precision, although he only published 15 digits of his findings.

In the 1700's, Euler discovered a number of new formulae for π, including

$$\frac{\pi^2}{6} = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2} + \dots$$

One incentive for computations of π during that time was to see whether or not the decimal expansion of π repeated (although very few people actually believed that π was rational). The question was finally resolved in the late 1700's when Lambert and Legendre proved the irrationality of π. In 1882, Lindemann proved that π was also transcendental, which answered, in the negative, the ancient Greek question of whether or not the circle could be squared with ruler and a pair of compasses[6].

In the twentieth century, π was computed to thousands and then millions of digits (for more details on the history of π up to about 1970, see [8]). It is rather surprising that until the 1970's, all computer computations of π still applied classical formulae, usually some variation of Machin's formula. Some remarkable new formulae were discovered by the Indian mathematician Ramanujan around 1910, but they remained unknown until quite recently when his writings were published.

---

[5]Later, it was found that this computation was in error after the 527$^{\text{th}}$ decimal place.

[6]This is because numbers constructible with ruler and compasses are necessarily algebraic.

Let us compute the constant π, using the following Ramanujan's formula[7]

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!\,(1103 + 26390k)}{(k!)^4\, 396^{4k}}. \tag{51}$$

In doing so, we first apply the square rooting, multiplication and division algorithms to compute the constant

$$\frac{2\sqrt{2}}{9801} = (-4, [3, \overline{1}, \overline{2}, 6, \overline{2}, 5, 6, \overline{4}, 5, 2, 2, 2, 5, 5, \overline{2}, \overline{3}, 1, \overline{1}, \ldots]), \tag{52}$$

and then use the methods of Section 2.5.3 to calculate the sum of the series in (51):

$$\begin{aligned}
a_0 &= (3, [1, 1, 0, 3]) \\
a_1 &= (-5, [3, \overline{3}, \overline{2}, 3, 2, 0, \overline{3}, 4, 3, 5, \overline{1}, 1, 2, 5, 3, 1, \overline{1}, \ldots]) \\
a_2 &= (-13, [2, 2, 4, 5, 4, \overline{2}, 5, 0, 2, 0, 1, 1, 4, \overline{3}, \overline{4}, 4, 4, \ldots]) \\
a_3 &= (-21, [2, 0, \overline{1}, 5, 1, \overline{3}, 5, 0, \overline{1}, 4, 5, \overline{1}, 6, \overline{1}, 0, \overline{3}, \overline{3}, \ldots])
\end{aligned} \tag{53}$$

The sum of the Ramanujan series is therefore

$$(3, [\underbrace{1, 1, 0, 3, 0, 0, 0, 0}_{8},\ \underbrace{3, \overline{3}, \overline{2}, 3, 2, 0, \overline{3}, 4}_{8},\ \underbrace{5, 7, 3, 4, 6, 3, 8, 1}_{8}\ \cdots]), \tag{54}$$

which, multiplied by (52), gives us

$$\frac{1}{\pi} = (-1, [3, 2, \overline{2}, 3, 1, 0, \overline{1}, \overline{1}, \overline{4}, 2, 2, \overline{2}, 4, \overline{2}, \overline{1}, 1, \overline{3}, \ldots]),$$

and taking the reciprocal, we have

$$\pi = (0, [3, 1, 4, 1, 6, \overline{1}, 3, \overline{4}, 5, 3, 6, \overline{1}, 0, \overline{2}, \overline{1}, 3, 2, 4, \ldots])$$

It can be seen from (53) and (54) that the difference in the exponents of

[7] In 1985, Gosper used this formula to compute 17 million digits of π.

adjacent terms is always 8, and so each term of this series produces an additional *eight* correct digits of π. This is considerably more efficient that the classical formulae, although the number of terms one must compute still increases *linearly* with the number of digits desired in the result, i.e. to compute π to twice as many digits, one must evaluate twice as many terms of the series.

A recent computation (1994) by the Chudnovsky brothers of Columbia University produced over four billion digits of π using the following infinite series resembling the above Ramanujan series

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409 + 545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}},$$

each term of which produces an additional 14 correct digits. There are also a number of high-order convergent algorithms, such as the Salamin-Brent algorithm [70, 14] which converges *quadratically* to π, or the Borwein algorithm that converges *quartically* or even *nonically* to π [13]. It is interesting that in our implementation Ramanujan's formula outperforms all these nominally faster algorithms.

Finally, we would like to mention the following novel scheme for computing *individual* hexadecimal digits of π that was recently discovered by Borwein, Bailey and Plouffe [7]. The scheme is based on the following extraordinary new formula for π:

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right).$$

Using a variant of the binary exponentiation algorithm, Borwein's formula can be used to compute individual hexadecimal digits of π at position $n + 1$ from the fractional part of $16^n \pi$ without computing the previous digits. Sample implementations of this method in Fortran and C are available from the web site `http://www.cecm.sfu.ca/personal/pborwein/`. Surprisingly, no similar formula has so far been found for the *decimal* expansion of π, although there is also no proof that a similar decimal scheme for π cannot exist.

Trigonometric functions

The circular functions sine $\sin x$ and cosine $\cos x$ are defined everywhere in $\mathbb{R}$ (or, in fact, $\mathbb{C}$), and are everywhere continuous and bounded. The tangent function

$$\tan x = \frac{\sin x}{\cos x}$$

is defined and is continuous everywhere on the real axis, except at the points

$$x = \pi(n + \frac{1}{2}), \quad n \in \mathbb{Z}.$$

Similarly, the cotangent function $\cot x = (\tan x)^{-1}$ is defined and continuous everywhere in $\mathbb{R}$, except at the points

$$x = \pi n, \quad n \in \mathbb{Z}.$$

The functions secant $\sec x$ and cosecant $\operatorname{cosec} x$ are defined as the reciprocals of $\cos x$ and $\sin x$ respectively, i.e.

$$\sec x = \frac{1}{\cos x}, \quad \operatorname{cosec} x = \frac{1}{\sin x}.$$

The trigonometric functions are periodic: $\sin x$ and $\cos x$ have a period of 2π, while $\tan x$ and $\cot x$ have a period of π. The function $\cos x$ is an even function, but $\sin x$, $\tan x$ and $\cot x$ are odd functions:

$$\begin{aligned} \sin x &= \operatorname{sgn} x \cdot \sin |x|, & \cos x &= \cos |x|, \\ \tan x &= \operatorname{sgn} x \cdot \tan x, & \cot x &= \operatorname{sgn} x \cdot \cot |x|. \end{aligned}$$

The functions sine and cosine have power series expansions (43) and (44), which converge reasonably fast for small x. Therefore, to compute $\sin x$ and $\cos x$ for arbitrary x, we must be able to reduce the argument range to a smaller one. Reduction of $\sin x$ to the interval $(-\pi/2, \pi/2)$ can be performed using the following formula which holds for arbitrary $x \in \mathbb{R}$: if

$$|x| = 2\pi\alpha + \pi\beta + \frac{\pi}{2}\gamma + \frac{\pi}{2}\delta,$$

where $\alpha \in \mathbb{N}_0$ is the integer number of periods, $\beta \in \{0,1\}$, $\gamma \in \{0,1\}$, $0 \leq \delta \leq 1$, then

$$\sin x = \sin \frac{\pi}{2} t,$$

where

$$t = (-1)^{\beta} \cdot \operatorname{sgn} x \cdot (\gamma + (-1)^{\gamma}\delta), \quad |t| < 1.$$

Alternatively, we can use the following formula which also holds for arbitrary $x \in \mathbb{R}$:

$$\sin \frac{\pi}{2} x = \sin \frac{\pi}{2} y,$$

where

$$y = \left| \left| \left\{ \frac{x-1}{4} \right\} \right| 4 - 2 \right| - 1, \quad |y| \leq 1.$$

In particular, if $|x| < 2$, we have $y = ||x-1| - 2| - 1$; if $|x| < 4$, then $y = 1 - |||x-1| - 2| - 2|$.

Similar formulae exist for other trigonometric functions; for example, $\tan x$ can be reduced to the interval $(-\pi/2, \pi/2)$ using the formula $\tan x = \tan y$, where

$$y = \pi \left\{ \frac{x}{\pi} + \frac{1}{2} \right\} - \frac{\pi}{2}.$$

It is important to note that even though the formulae quoted are *discontinuous*, they represent continuous functions. Range reduction is used in the algorithms merely to speed up convergence, and it is not critical that it should be *always* performed accurately, as long as it is *mostly* accurate. A non-deterministic conditional could be used to perform range reduction, and as long as the function being evaluated is continuous, a false result can never lead to a non-convergent series – at worst, the series would converge more slowly.

Example 8. *Compute* $\sin 1000$ *in radix* $r = 10$, $\rho = 6$.

The essential idea of the method described above is to compute the argument modulo 2π, and use the power series (43) with the new argument:

$$\begin{aligned}
x = 1000 = (3,[1]), \quad \left\lfloor \frac{x}{2\pi} \right\rfloor &= 159 = (2,[1,5,9]),\\
x' = x - \left\lfloor \frac{x}{2\pi} \right\rfloor \cdot 2\pi &= (0,[1,0,\overline{3},4,\overline{5},4,\overline{4},2,\overline{4},\overline{2},4,4,\ldots]),\\
\sin x = \sin x' &= \sum_{n=0}^{\infty}(-1)^n \frac{(x')^{2n+1}}{(2n+1)!}
\end{aligned}$$

Computing the terms of the series, we have

$$\begin{aligned}
a_0 &= (0,[1,0,\overline{3},3,5,4,\overline{4},1,6,\overline{2},4,4,6,\overline{3},5,0,2,\ldots])\\
a_1 &= (-1,[\overline{1},\overline{5},\overline{4},2,2,\overline{2},2,\overline{2},\overline{4},\overline{4},\overline{2},1,\overline{1},0,2,1,\ldots])\\
a_2 &= (-2,[1,\overline{3},3,\overline{1},\overline{3},5,1,0,4,\overline{2},\overline{4},4,3,\overline{4},6,\overline{1},\ldots])\\
a_3 &= (-4,[\overline{2},4,\overline{4},\overline{4},\overline{5},0,\overline{1},3,\overline{2},\overline{2},\overline{5},0,1,1,0,0,\ldots])\\
&\ldots\\
\sin x &= (0,[1,\overline{2},3,\overline{3},\overline{1},\overline{2},0,\overline{4},\overline{6},0,5,3,2,0,0,3,\overline{4},\overline{4},\ldots])\\
&= 0.82687954053200256\ldots
\end{aligned}$$

The method of determining the sum of infinite series described in Section 2.5.3 can also be successfully applied to other functions, such as the inverse trigonometric or hyperbolic functions, which have reasonably fast convergent power series expansions, e.g.

$$\arcsin x = \sum_{k=0}^{\infty} \frac{(2k)!}{2^{2k}(k!)^2(2k+1)} x^{2k+1}, \qquad |x| < 1$$

$$\arctan x = \frac{x}{1+x^2} \sum_{k=0}^{\infty} \frac{2^{2k}(k!)^2}{(2k+1)!} \left(\frac{x^2}{1+x^2}\right)^k, \qquad |x| < \infty$$

$$\sinh x = \sum_{k=1}^{\infty} \frac{x^{2k+1}}{(2k+1)!}$$

$$\cosh x = \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!}$$

2.6 Conversion algorithms

Although the algorithms we have discussed so far can work in an arbitrary positional radix-r system, a human user of an exact arithmetic package is likely to want to input numbers in decimal form, and expect results to be output as decimals.

In this section we shall discuss the online conversion of numbers from redundant positional notation with one radix into redundant positional notation with another radix, including, but not limited to, decimal conversion. We give offline algorithms for conversion of redundant decimal numbers into conventional form and vice versa. In view of the fact that there is no *online* algorithm for converting a redundant number into non-redundant form, we conclude by introducing a variant of the redundant representation, which is in many ways similar to the conventional non-redundant notation.

2.6.1 Redundant radix conversion

We assume that a positional system is identified by the two parameters (r, ρ), where r is the radix, and ρ is the range parameter as per Section 2.2. Suppose that we are converting from radix r_1 to radix r_2, i.e. given a number

$$a = (e_a, (a_0, a_1, a_2, \ldots, a_n, \ldots)_{r_1}), \quad |a_k| \leq \rho_1, \tag{55}$$

we would like to find its (r_2, ρ_2) representation

$$a' = (e'_a, (a'_0, a'_1, a'_2, \ldots, a'_n, \ldots)_{r_2}), \quad |a'_k| \leq \rho_2. \tag{56}$$

To avoid dealing with exponents in representations (55) and (56), we shall assume $e_a = e'_a = 0$ and allow the first entries a_0 and a'_0 to be unbounded. By doing so, we can set $a'_0 = a_0$ and convert the fractional part of (55) only. For example, if we wanted to convert $(3, [6, 3, \overline{5}, 1])_{10}$ from radix 10 to radix 16, we would write

$$(3, [6, 3, \overline{5}, 1])_{10} = (0, [625, 1])_{10}$$

and convert only the fractional part 0.1_{10} of the number, observing that the unnormalized integer part 625 remains the same in any radix. Since the integer part of a number is always finite, normalizing it in radix r_2 is trivial and can be done using conventional methods of radix conversion (see e.g. [45]).

In other words, equations (55) and (56) can be rewritten in the form

$$a = a'_0 + a'_1 r_2^{-1} + \ldots + a'_n r_2^{-n} + \ldots = a_0 + a_1 r_1^{-1} + \ldots + a_n r_1^{-n} + \ldots,$$

from which it is clear that $a'_0 = a_0$, and a'_1 is the integer part of the right-hand side of

$$a'_1 + a'_2 r_2^{-1} + \ldots + a'_n r_2^{-n+1} + \ldots = a_1 r_2 r_1^{-1} + a_2 r_2 r_1^{-2} + \ldots + a_n r_2 r_1^{-n} + \ldots \tag{57}$$

Since $|a_i r_2| \leq \rho_1 r_2$ for all $i \in \mathbb{N}$, the sequence on the right-hand side of (57) can be reduced to a sequence

$$a_0^{(1)} + a_1^{(1)} r_1^{-1} + a_2^{(1)} r_1^{-2} + \ldots + a_n^{(1)} r_1^{-n} + \ldots$$

with $\left|a_k^{(1)}\right| \leq \rho_1$ $(k \in \mathbb{N})$ in $\lceil \log_{r_1}(\rho_1 r_2 - \rho_1 + 1) \rceil$ applications of `reduce`. Thus, $a'_1 = a_0^{(1)}$ and we have

$$a'_2 + a'_3 r_2^{-1} + \ldots + a'_n r_2^{-n+2} + \ldots = a_1^{(1)} r_2 r_1^{-1} + a_2^{(1)} r_2 r_1^{-2} + \ldots + a_n^{(1)} r_2 r_1^{-n} + \ldots,$$

an equation similar to (57). Again, the coefficients $a_i^{(1)} r_2$ satisfy $\left|a_i^{(1)} r_2\right| \leq \rho_1 r_2$ for all $i \in \mathbb{N}$, and we can reduce the right-hand side to

$$a_0^{(2)} + a_1^{(2)} r_1^{-1} + a_2^{(2)} r_1^{-2} + \ldots + a_n^{(2)} r_1^{-n} + \ldots,$$

where all numbers $a_k^{(2)}$ (except perhaps $a_0^{(2)}$) are bounded by ρ_1. Setting $a'_2 = a_0^{(2)}$ and continuing in this fashion, we obtain the sequence

$$(a'_0, a'_1, a'_2, \ldots, a'_n, \ldots)_{r_2} = (a_0, a_0^{(1)}, a_0^{(2)}, \ldots, a_0^{(n)}, \ldots)_{r_2}, \tag{58}$$

which represents a in radix r_2. Notice that this sequence is not necessarily canonical, as the numbers $a_0^{(k)}$ are obtained through normalization of $a_1^{(k-1)}$, hence

$$\left|a_0^{(k)}\right| \leq \left\lfloor \left|a_1^{(k-1)} r_2 r_1^{-1}\right| \right\rfloor \leq \left\lfloor \rho_1 r_2 r_1^{-1} \right\rfloor.$$

The final result is therefore obtained by normalizing (58) $\lceil \log_{r_2}(\lfloor \rho_1 r_2 r_1^{-1} \rfloor - \rho_2 + 1) \rceil$ times.

A special case of the above algorithm arises when $r_2 = r_1^k$: it is easy to see that

$$(a_0, a_1, a_2, \ldots, a_n, \ldots)_{r_1} = (a'_0, a'_1, a'_2, \ldots, a'_m, \ldots)_{r_2},$$

where

$$a'_0 = a_0, \quad a'_i = (a_{ik-k+1}, \ldots, a_{ik-1}, a_{ik})_{r_1}.$$

Therefore we have a very simple technique for converting between, say, radix 10 and 1000.

Example 9. *Convert π from radix 10 to radix 16 (assuming $\rho_1 = 6$, $\rho_2 = 9$).*

The value of π *in radix* 10 *was computed in Section 2.5.4:*

$$\pi = (0, [3, 1, 4, 1, 6, \overline{1}, 3, \overline{4}, 5, 3, 6, \overline{1}, 0, \overline{2}, \overline{1}, 3, \ldots])_{10}.$$

We already know that $a'_0 = 3$*, and compute* a'_1 *by normalizing*

$$[0, 16, 64, 16, 96, -16, 48, -64, 80, 48, 96, -16, 0, -32, -16, 48, \ldots]_{10}$$

in radix 10 *twice:*

$$[2, 2, 6, 6, \overline{6}, 9, \overline{8}, 4, 5, 8, \overline{6}, 4, \overline{3}, \overline{4}, 9, 1, \ldots]_{10} = [2, 3, \overline{3}, \overline{5}, 5, \overline{2}, 2, 4, 6, \overline{3}, 4, 4, \overline{3}, \overline{3}, \overline{1}, 2, \ldots]_{10}.$$

The first entry of the last sequence is $a'_1 = 2$*; the next entry* a'_2 *is calculated by double-normalizing*

$$[0, 48, -48, -80, 80, -32, 32, 64, 96, -48, 64, 64, -48, -48, -16, 32, \ldots]_{10}$$

in radix 10*:*

$$[5, \overline{7}, \overline{6}, 8, \overline{3}, 1, 8, 14, \overline{9}, 8, 10, \overline{1}, \overline{3}, 0, 7, \overline{1}, \ldots]_{10} = [4, 2, 5, \overline{2}, \overline{3}, 2, \overline{1}, 3, 2, \overline{1}, 0, \overline{1}, \overline{3}, 1, \overline{3}, \overline{1}, \ldots]_{10}.$$

Thence we deduce $a'_2 = 4$*, and proceeding recursively, we compute*

$$\pi = [3, 2, 4, 4, \overline{1}, 7, \overline{5}, \overline{8}, 8, 8, 6, \overline{6}, 3, 0, 9, \overline{3}, 3, 1, 3, \ldots]_{16}.$$

2.6.2 Offline conversion to standard representation

In this section we show how to convert a number from the redundant radix-r form to the standard decimal notation. For reasons explained earlier (see Section 1.5.4), such conversion cannot be performed online; therefore, a *precision* (the number of decimal digits required in the final result) must be specified in advance.

Suppose the number a to be converted is represented in radix r by the sequence

$(e'_a, (a'_k)_{k\in\mathbb{N}_0})_r$. Using the results of Section 2.6.1, we first convert the number into redundant *decimal* form:

$$a = 10^{e_a} \cdot (a_0 + a_1 10^{-1} + a_2 10^{-2} + \ldots + a_n 10^{-n} + \ldots).$$

If precision $n \in \mathbb{N}$ is required, we truncate the number to

$$a = 10^{e_a} \cdot (a_0 + a_1 10^{-1} + a_2 10^{-2} + \ldots + a_m 10^{-m}), \quad m = e_a + n + 1 \tag{59}$$

if $e_a + n \geq 0$, and $a = 10^{e_a} \cdot 0$ otherwise. The latter comparison is needed to avoid unnecessary computation — for example, if someone requests $2 \cdot 10^{-100}$ to 5 digits' precision, they should get 0 as the result and not $2.0000 \cdot 10^{-100}$.

The next step is to determine the sign of the finite number, print "$-$" if it is negative and negate each entry in the sequence to make it positive. In doing so, we can also cancel all leading zeros in (59), which we could not do before as the cancellation of zeros in an infinite number is a possibly non-terminating process.

The conversion of a positive finite number into standard decimal form consists of two main steps:

1) applying the function $f_{i_1\cdots i_m}$ (see Definition 8) to reduce the number of negative terms in the sequence (left to right), and

2) propagating the carries from the least significant digit to the most significant one (right to left) to eliminate the remaining negative terms.

Let us consider these steps in more detail for a given number $(a_0, a_1, \ldots, a_m)_{10}$, where $a_0 > 0$. Let

$$(a'_0, a'_1, \ldots, a'_m)_{10} = f_{i_1\cdots i_m}((a_0, a_1, \ldots, a_m)_{10}), \tag{60}$$

where

$$i_k = \begin{cases} -k, & \text{if } a_k < 0 \\ 0, & \text{otherwise} \end{cases}$$

This means that if $a_1 < 0$, we are replacing $(a_0, a_1, a_2, \ldots, a_m)_{10}$ with $(a_0 - 1, a_1 + 10, a_2, \ldots, a_m)_{10}$, if $a_2 < 0$, we are replacing the last sequence by $(a_0 - 1, a_1 + 9, a_2 + 10, \ldots, a_m)_{10}$, etc. The resulting sequence $(a_0', a_1', \ldots, a_m')_{10}$ does not necessarily have all positive entries, e.g.

$$\begin{aligned} f_{0,-2,0,-4,-5}((2,0,\overline{5},0,\overline{1},\overline{6})_{10}) &= f_{-5}(f_{-4}(f_{-2}((2,0,\overline{5},0,\overline{1},\overline{6})_{10}))) \\ &= f_{-5}(f_{-4}((2,\overline{1},5,0,\overline{1},\overline{6})_{10})) \\ &= f_{-5}((2,\overline{1},5,\overline{1},9,\overline{6})_{10}) \\ &= (2,\overline{1},5,\overline{1},8,4)_{10}. \end{aligned}$$

To weed out the remaining (-1)'s in the sequence, we need to propagate the carries from right to left, replacing (-1)'s by 9's, and adjusting the adjacent entries:

$$(2,\overline{1},5,\overline{1},8,4)_{10} = (2,\overline{1},4,9,8,4)_{10} = (1,9,4,9,8,4)_{10}.$$

2.6.3 Online conversion to "almost standard" representation

Although offline conversion can be used to convert the final result to the conventional form, it is sometimes desirable to inspect intermediate results without performing such conversion (as offline conversion always implies loss of precision). In this section we introduce an "almost standard" representation of exact real numbers which can be computed online and understood more easily than the usual redundant $-\rho$-to-ρ representation.

Definition 11. *The mantissa $M = (a_n)_{n\in\mathbb{N}_0}$ of a real number $x = (r, \rho, E, M)$ is said to be "almost standard", if for all $n \in \mathbb{N}_0$,*

$$a_n \in \left\{\overline{1}, 0, 1, \ldots, r-1\right\} \tag{61}$$

for $x > 0$, or

$$a_n \in \left\{\overline{r-1}, \ldots, \overline{1}, 0, 1\right\} \tag{62}$$

for $x < 0$. (If $x = 0$, then by definition $a_n = 0$.)

In other words, the almost standard notation uses a non-symmetrical digit set, either (61) or (62), depending on the sign of the number to be represented. When the number being represented is positive and $r = 10$, the "almost standard" notation is indeed very similar to the conventional decimal representation, the only difference being the appearance of $\overline{1}$'s in its expansion. For example, $0.854\overline{1}734222\ldots$ is an almost standard representation of $1.\overline{2}540\overline{3}34223\overline{3}\ldots$, and $\overline{2}.\overline{69}0\overline{5}1\overline{5}\overline{12}1\overline{83}\ldots = -2.6905\overline{1}512\overline{1}83\ldots$ is an almost standard representation of $\overline{3}.31\overline{1}505\overline{12}02\overline{3}\ldots$

Next we shall show that there exists an online algorithm that converts any redundant radix-r number to almost standard form. Since the almost standard representation of a negated number is obtained by negating all individual digits, we shall only discuss the conversion of positive numbers.

The algorithm for we shall put forward uses the equivalence functions $f_{(i_n)}$ introduced by Definition 9. If $(a_0, a_1, \ldots, a_n, \ldots)_r$ is the number to be converted, then similarly to (60), we define

$$(a'_0, a'_1, \ldots, a'_n, \ldots)_r = f_{(i_k)_{k\in\mathbb{N}}}((a_0, a_1, \ldots, a_n, \ldots)_r),$$

where

$$i_k = \begin{cases} -k, & \text{if } a_k < 0 \\ 0, & \text{otherwise} \end{cases}$$

It is clear that the sequence $(a'_n)_{n\in\mathbb{N}_0}$ cannot contain any negative digits, apart perhaps from (-1)'s. Indeed, the n-th element of the sequence can only be changed by f_{-n} and $f_{-(n+1)}$ — if $a_n < 0$, it becomes positive after the application of f_{-n}; if $a_n \geq 0$, it is left intact by f_{-n} but may be decreased by 1 by $f_{-(n+1)}$ if $a_{n+1} < 0$; hence, $a_n \geq -1$.

Chapter 3

Exact complex arithmetic

3.1 Introduction

In most modern digital computers, complex numbers are represented as pairs of real numbers, and arithmetic operations on them are developed in terms of the corresponding operations on reals. The need to maintain separate representations for the real and imaginary parts of complex numbers makes them much more awkward to compute than the reals. For instance, complex addition or subtraction involves two real additions or subtractions, while multiplication of two complex numbers involves four real multiplications, a real addition, and a real subtraction:

$$(a + ib)(c + id) = (ac - bd) + i(bc + ad).$$

If multiplication is a much slower operation than addition, the above formula can be improved upon by using the relation

$$(a + ib)(c + id) = (ac - bd) + i[(a + b)(c + d) - ac - bd],$$

which involves only three real multiplications (ac, bd, $(a + c)(b + d)$), two real additions and three real subtractions [56]. Complex division in this representation, however, is inevitably even more "complex":

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}.$$

In this chapter, we introduce a method of performing exact complex arithmetic that does not involve separating the complex numbers into their real and imaginary parts. This is achieved by using the representation of complex numbers in positional notation using a complex (in fact, imaginary) base ri, with integer digits [40]. We show that addition, subtraction, multiplication and division can be performed directly in this notation similarly to exact real arithmetic in radix r^2, the only difference being the normalization procedure. Benchmark tests have shown that the unified representation performs better than the conventional one (see Chapter 4).

3.1.1 Representations of complex numbers

Historically speaking, the complex numbers originated from the desire for a symbolic representation for the solutions of such equations as $x^2 + 1 = 0$, otherwise irreducible over $\mathbb{R}$. In modern terminology, we would say that the field of complex numbers $\mathbb{C}$ was a finite algebraic extension of $\mathbb{R}$ of degree 2, $\mathbb{C} = \mathbb{R}\left(\sqrt{-1}\right)$, which by mere coincidence also happened to be the *algebraic closure* of $\mathbb{R}$ (see e.g. [47]). The latter fact is precisely the famous Fundamental Theorem of Algebra[1], and the exception rather than the rule. As well as algebraically closed, $\mathbb{C}$ also turns out to be *complete* with respect to the norm that extends the norm $|\cdot|$ on $\mathbb{R}$, which makes it the convenient number system, as it is, in which to study calculus and analysis. The use of complex numbers in analysis gave a completeness and insight that had previously been lacking, and they are now so widely used in all areas of mathematics that they are accepted without question.

The definition of the complex numbers historically came before the rigorous

[1]The Fundamental Theorem of Algebra (FTA) states that *every polynomial equation with complex coefficients has a complex root.*

definition of the real numbers in terms of Cauchy sequences. In an apparent attempt to put things in logical order, it became conventional to discuss complex numbers in terms of the real number pairs used to represent them. The subsequent discussion is predicated on the assumption that the reader is acquainted with this usual representation of complex numbers. We shall therefore only briefly review some concepts and facts (for a complete treatment, see e.g. Lang's *Complex Analysis* [48]).

Conventional representation

We take the conventional form of a complex number to be

$$z = x + iy,$$

where the numbers $x, y \in \mathbb{R}$ are referred to as the *real* and *imaginary* parts of the complex number z and denoted by $\operatorname{Re} c$ and $\operatorname{Im} c$ respectively.

The *modulus* of a complex number $z = x + iy$ will be denoted by $|z|$ and defined by

$$|z| = \left(x^2 + y^2\right)^{1/2}.$$

The *sum*, *difference* and *product* of the two numbers $z_1 = x_1 + iy_1$ and $z_2 = x_2 + iy_2$ are defined by

$$z_1 \pm z_2 \stackrel{def}{=} (x_1 \pm x_2) + i\,(y_1 \pm y_2)\,,$$

and

$$z_1 \cdot z_2 \stackrel{def}{=} (x_1 x_2 - y_1 y_2) + i\,(x_1 y_2 + x_2 y_1)$$

(the latter is exactly the result of a formal application of the distributive law to $(x_1 + iy_1)\,(x_2 + iy_2)$).

The *complex conjugate* of $z = x + iy$ will be denoted by $z^* \stackrel{def}{=} x - iy$. Note that the product $z \cdot z^*$ is always real, which gives us a convenient way of calculating

z_1/z_2 by multiplying both numerator and denominator by z_2^*; thus

$$\frac{z_1}{z_2} = \frac{x_1x_2 + y_1y_2}{x_2^2 + y_2^2} + i\frac{y_1x_2 - x_1y_2}{x_2^2 + y_2^2}.$$

This formula uses only real numbers and is normally used as the definition of *division* for complex numbers.

Single-component representation

On account of the complexity of even the four basic arithmetic operations, several proposals have been made for a more concise, single-component representation of the complex numbers, which would allow complex arithmetic to be done in a fairly unified manner, relieving the programmer of the burden of treating the real and imaginary parts separately [43, 61, 58, 37]. The essence of these proposals is to choose the radix to be a complex or an imaginary number, and then use sequences of *real* digits to express the complex quantity as a weighted sum of powers of that base.

In this chapter we shall consider the use of the number ri as a radix. For this system, we assume a positional notation

$$z_0.z_1z_2\cdots z_n\cdots = \sum_{k=0}^{\infty} z_k \cdot (ri)^{-k}, \tag{63}$$

where the weights associated with position $k \in \mathbb{N}_0$ are $(ri)^{-k}$:

$$\begin{array}{ccccccccc} k & : & 0 & -1 & -2 & -3 & -4 & -5 & -6 & \cdots \\ (ri)^{-k} & : & 1 & -r^{-1}i & -r^{-2} & r^{-3}i & r^{-4} & -r^{-5}i & -r^{-6} & \cdots \end{array}$$

As can be seen from the above, the even-numbered positions in this expansion all have real weights, while the odd-numbered positions all have imaginary weights. The real weights are related to each other as in base $(-r^2)$, and so are the imaginary ones after factoring out the common ri:

Even-numbered positions:

$$\begin{array}{ccccccc} k & : & 0 & -2 & -4 & -6 & \cdots \\ (ri)^{-k} & : & 1 & -r^{-2} & r^{-4} & -r^{-6} & \cdots \end{array} \tag{64}$$

Odd-numbered positions (after factoring out ri):

$$\begin{array}{ccccccc} k & : & -1 & -3 & -5 & \cdots & \\ (ri)^{-k} & : & -r^{-2} & r^{-4} & -r^{-6} & \cdots & \times ri \end{array} \tag{65}$$

For example, let $r = 10$ and the number z be represented by

$$z = 19\,.\,36\;23\;40$$

Then

$$\begin{aligned} z &= 19 - 36 \cdot 10^{-1}\, i - 23 \cdot 10^{-2} + 40 \cdot 10^{-3}\, i \\ &= 18.77 - 3.56\, i \end{aligned}$$

Thus, in order to express a complex number $z = x + iy$ in this format, we first write its real part x as an expansion in negative radix $(-r^2)$, then we write a radix-$(-r^2)$ expansion of ry, and finally we interleave the two parts to obtain a single-component expansion.

Negative-radix systems have been extensively described in the literature [60, 88, 78, 23, 72, 2], and the rules for addition, negation, multiplication and division of finite numbers in these systems are obtained by straightforward modification of the standard rules. At least one computer has been built based on such a system [59]. However, the only advantage of negative-radix number systems seems to reside in the fact that all finite real numbers, whether positive or negative, can be represented in these systems without a separate sign digit, thus obviating the need for special treatment of negative numbers [78].

Even though the idea of absorbing the sign into the number representation may have a moderate practical advantage in fixed-size-format number systems, where only a finite precision is available for representing numbers, the change-over from finite to infinite and from positive to signed-digit strings renders this method indistinguishable from the conventional positive-radix method. Indeed, if we have a signed-digit expansion in a radix $r > 0$,

$$x = \sum_{k=0}^{\infty} x_k r^{-k}, \ -\rho \leq x_k \leq \rho,$$

we can easily transform it into the corresponding negative-radix expansion

$$x = \sum_{k=0}^{\infty} x'_k \left(-r\right)^{-k}, \ -\rho \leq x'_k \leq \rho,$$

where

$$x'_k = \left(-1\right)^k x_k, \ k \in \mathbb{N}_0,$$

and vice versa. Thus, the use of a symmetrical signed-digit set blurs the distinction between positive- and negative-radix number systems and allows us to circumvent the difficulties of coping with alternating signs in negative-radix expansions.

Let us consider a radix-ri expansion

$$\sum_{k=0}^{\infty} z_k \cdot (ri)^{-k} \tag{66}$$

of a complex number z with

$$z_k \in \mathbb{R}, \ |z_k| \leq \rho, \ \rho \in \left[\frac{r^2}{2}, \ r^2 - 1\right]. \tag{67}$$

It is clear that only a subset of the complex numbers can be represented by such sequences; namely, the numbers $z = x + iy$ with

$$|x| \leq \rho \frac{r^2}{r^2 - 1}, \ |y| \leq \rho \frac{r}{r^2 - 1}. \tag{68}$$

Note that the representation (66) of a complex number $z = x + iy$ is asymmetrical with respect to x and y because of the presence of the factor $1/r$ in the expansion of y:

$$\begin{aligned} x &= z_0 - z_2 r^{-2} + z_4 r^{-4} + \ldots \\ y &= \frac{1}{r}\left(-z_1 + z_3 r^{-2} - z_5 r^{-4} + \ldots\right) \end{aligned} \tag{69}$$

How can we represent *all* complex numbers in the form (66)? The standard solution to this problem is to *scale* any complex number onto the above range (68). The scaling could in principle be applied separately to the real and imaginary parts of the complex number, but the arithmetic operations become much simpler when the real and imaginary parts share a *common* exponent [91].

Exponent representation

The exponent representation of complex numbers is similar to the commonly used floating-point representation of real numbers and consists of two main parts: the exponent $E \in \mathbb{Z}$, and the mantissa M, which is a sequence of integers $(z_n)_{n=0}^{\infty} \in \mathbb{Z}$. The complex number z, represented by the pair (E, M), has the value

$$z = B^E \cdot \sum_{n=0}^{\infty} z_n \, (ri)^{-n}, \tag{70}$$

where B is the *base* of the exponent (usually a positive integer).

We shall now show that *all* complex numbers can be represented in a radix-ri system as an expansion of the form (70), where all z_n are in the range (67). Note that for the time being, we are interested primarily in the general properties of this system, and do not concern ourselves with the questions of computability or effective convergence of the series in (70).

Let $Z \in \mathbb{C}$ be an arbitrary complex number, $Z = X + iY$, $X, Y \in \mathbb{R}$. We aim to find an exponent $E \in \mathbb{Z}$ and two numbers $x, y \in \mathbb{R}$ satisfying condition (68) such that

$$Z = B^E \cdot (x + iy)\,. \tag{71}$$

From (71) and (68), we find that

$$B^E \geq \frac{r^2-1}{\rho r^2}\,|X|\,,\ B^E \geq \frac{r^2-1}{\rho r}\,|Y|\,,$$

and therefore,

$$B^E \geq \max\left(\frac{r^2-1}{\rho r^2}\,|X|\,,\ \frac{r^2-1}{\rho r}\,|Y|\right). \tag{72}$$

The best value of E in (72) is the minimal one, because it corresponds to the largest possible values of x and y in (71) that satisfy (68). Thus, we choose the exponent

$$E = \left\lceil \log_B \max\left(\frac{r^2-1}{\rho r^2}\,|X|\,,\ \frac{r^2-1}{\rho r}\,|Y|\right)\right\rceil.$$

3.2 Redundant radix-ri exponent representation of complex numbers

The main decision to be made regarding the exponent representation is the choice of base B. The choice partially depends on the way numbers are generated in the system, and the usual practice is to choose the exponent base to be the same as the main radix. We can assume that most of the complex data will be fed into the system in the conventional form, with the real and imaginary parts separately represented in real radix r', where r' is either r or r^2. Leaving negative bases such as $(-r)$ or $(-r^2)$ out of consideration, we have only two practicable choices for the exponent base — r and r^2. In either case, the real number pairs first have to be converted to $(-r^2)$, and then interleaved to obtain a radix-ri expansion (conversion from and to radix-ri is discussed in Section 3.5). The particular choice depends on how the trade-off between addition and conversion is to be made — as we shall see, exponent base r simplifies conversion at the expense of addition, while r^2 streamlines addition to the detriment of conversion. We naturally regard addition as more important than conversion; after all, conversion should only be performed twice — once at the beginning and once at the end of a computation. Taking the aforesaid into account, we have chosen r^2 to be the exponent base.

To summarize the results of the previous section and the preceding discussion, we define a *computable exact complex number* z similarly to (Chapter 2) as a quadruple (R, ρ, E, M), where $R \in \mathbb{N}$, $R > 4$ is the squared imaginary part of the radix value ($R = r^2$), as well as the base of the signed *exponent* $E \in \mathbb{Z}$, the *range parameter* ρ is an integer with $R/2 < \rho < R-1$, and the *mantissa* M is an *effectively given* sequence of numbers $(z_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$ such that

$$|z_n| \leq Cn, \ n \in \mathbb{N},$$

where $C > 0$ is a constant common to all complex numbers in a system. The value of $z = (R, \rho, E, (z_n)_{n\in\mathbb{N}_0})$ is taken as

$$z = R^E \cdot \sum_{n=0}^{\infty} z_n (ri)^{-n}.$$

3.3 Complex normalization

Normalization, in the context of radix-r redundant signed-digit positional weighted systems, refers to the process of restoring the individual digits of an effectively given sequence $(a_n)_{n\in\mathbb{N}_0} \in \mathbb{Z}$, to the canonical range $[-\rho, \rho]$, where ρ is an integral range parameter between $r/2$ and $r-1$. By this is meant the finding of a *normalized* sequence, $(a'_n)_{n\in\mathbb{N}} \in [-\rho, \rho]$ such that

$$\sum_{n=0}^{\infty} a'_n r^{-n} = \sum_{n=0}^{\infty} a_n r^{-n}.$$

Note that the first digit, a'_0, of thus normalized sequence may generally remain unbounded.

A *canonical* representation for the mantissa of a *complex* number z written to base ri is a sequence

$$(z_0, z_1, z_2, \cdots, z_n, \cdots)_{ri},$$

where $|z_n| < \rho$ for all $n \in \mathbb{N}_0$, and ρ is now a positive integer between $R/2$ and $R-1$ ($R = r^2$). Likewise, we define a *normalized* representation as one that has $|z_n| < \rho$ for all $n \in \mathbb{N}_0$, except perhaps $n = 0$ and 1.

Let $(z_i)_{i\in\mathbb{N}_0}$ be an unnormalized mantissa of a complex number $z = R^E \cdot \sum_{i=0}^{\infty} z_i (ri)^{-i}$. Recalling that

$$\sum_{i=0}^{\infty} z_i (ri)^{-i} = \sum_{k=0}^{\infty} (-1)^k z_{2k} R^{-k} + i \cdot \frac{1}{r} \sum_{k=0}^{\infty} (-1)^{k+1} z_{2k+1} R^{-k},$$

the problem of ri-normalizing $(z_n)_{n\in\mathbb{N}_0}$ is tantamount to normalizing two radix-R sequences $(x_n)_{n\in\mathbb{N}_0}$ and $(y_n)_{n\in\mathbb{N}_0}$, where

$$x_n = (-1)^n z_{2n}, \quad y_n = (-1)^{n+1} z_{2n+1}. \tag{73}$$

Once the sequences $(x_n)_{n\in\mathbb{N}_0}$ and $(y_n)_{n\in\mathbb{N}_0}$ have been normalized in radix R, we can once again alternate their signs in accordance with (73), and combine them back into a single sequence z_n.

Normalization of radix-R sequences has been discussed in Chapter 2.

3.4 Basic arithmetic operations

3.4.1 Addition and subtraction

Let $z = (E_z, (z_0, z_1, \cdots, z_n, \cdots)_{ri})$ and $w = (E_w, (w_0, w_1, \cdots, w_n, \cdots)_{ri})$ be the two normalized complex numbers to be added. If $E_z = E_w$, the procedure for addition is very straightforward: the digits of the two sequences are added and the resulting sequence

$$(z_0 + w_0,\ z_1 + w_1, \cdots,\ z_n + w_n, \cdots)_{ri}$$

is then normalized. The normalization can be done in a single pass, since

$$|z_n + w_n| \leq |z_n| + |w_n| \leq 2\rho \leq R + \rho - 1.$$

If the exponents of z and w are not equal, the mantissa of one of the operands has to be adjusted to make them equal. The shift has to be to the right to avoid the loss of the most significant digits — therefore, it is the mantissa of the addend with the *smaller* exponent that is shifted. Since addition is a commutative operation, we can assume that $e = E_z - E_w > 0$ without loss of generality. The details of the radix alignment are as follows:

$$w = R^{E_z} \cdot (w'_0, w'_1, \ldots, w'_n, \ldots)_{ri}\,,$$

where

$$(w'_n)_{n \in \mathbb{N}_0} = \begin{cases} (\underbrace{0, \cdots, 0}_{2e \text{ zeros}}, w_0, \cdots, w_n, \cdots)_{ri}, & \text{if } e \equiv 0 \bmod 2 \\ (\underbrace{0, \cdots, 0}_{2e \text{ zeros}}, -w_0, \cdots, -w_n, \cdots)_{ri}, & \text{if } e \equiv 1 \bmod 2 \end{cases} \tag{74}$$

The mantissa in (74) is left unchanged if the number of right shifts is divisible by 4, and negated otherwise. That such is the case is explained by the fact that two right-shifts are equivalent to dividing the mantissa by $(-r^2) = -R$, while four right-shifts are equivalent to dividing it by $r^4 = R^2$:

$$\begin{aligned} R^E \cdot (w_0, \cdots, w_n, \cdots)_{ri} &= R^{E+1} \cdot (0, 0, -w_0, \cdots, -w_n, \cdots)_{ri} \\ &= R^{E+2} \cdot (0, 0, 0, 0, w_0, \cdots, w_n, \cdots)_{ri}. \end{aligned}$$

3.4.2 Multiplication

Complex multiplication in radix ri is also a very straightforward modification of radix-r multiplication discussed in Section 2.4.2 of Chapter 2, and is performed by multiplying the formal series and renormalizing the result. Specifically, if $z =$

$R^{E_z} \cdot \sum_{k=0}^{\infty} z_k (ri)^{-k}$ and $w = R^{E_w} \cdot \sum_{m=0}^{\infty} w_m (ri)^{-m}$ are the numbers to be multiplied, we form their Cauchy product

$$\sum_{k=0}^{\infty} c_k (ri)^{-k} = \sum_{k=0}^{\infty} \left(\sum_{m=0}^{k} z_m w_{k-m} \right) (ri)^{-k},$$

where $c_k = \left(\sum_{m=0}^{k} z_m w_{k-m} \right)$, and proceed to ri-normalize the c_k in groups of n rows each (see Figures 2 and 3).

3.4.3 Division

If $z \in \mathbb{C}$ is the dividend and $w \in \mathbb{C}$, $w \neq 0$ the divisor, the algorithm for radix-ri system can be obtained from the classical one by using a recurrence relation similar to (25):

$$P_{n+1} = ri \cdot (P_n - q_n w), \; n \in \mathbb{N}_0,$$

where $P_0 = z$ and $(q_n)_{n \in \mathbb{N}_0}$ are the digits of the quotient $Q = z/w$. In perfect analogy with the real case, we infer that

$$P_n = (ri)^n \left[z - \left(q_0 + q_1 (ri)^{-1} + \ldots + q_{n-1} (ri)^{-n+1} \right) w \right], \; n \in \mathbb{N},$$

and so if Q is correct to n radix-ri digits, we have

$$\left| z - \left(q_0 + q_1 (ri)^{-1} + \ldots + q_{n-1} (ri)^{-n+1} \right) w \right| \leq \frac{|P_n|}{r^n}.$$

If q_n were selected in such a way that P_n would stay bounded, the above relation would guarantee convergence of the algorithm.

On the digit level, let z and w be represented by the following radix-ri sequences:

$$z = (z_0, z_1, \cdots, z_n, \cdots)_{ri}, \quad w = (w_0, w_1, \cdots, w_n, \cdots)_{ri},$$

where w is appropriately scaled so that either $w_0 \neq 0$ or $w_1 \neq 0$. To gain an impression of how to choose the digit selection function, we observe that

$$\begin{aligned}P_{n+1} &= ri \cdot (p_{n0} - q_n w_0,\, p_{n1} - q_n w_1,\, \cdots,\, p_{nk} - q_n w_k,\, \cdots)_{ri} \\ &= \left(p_{n1} - q_n w_1,\, -r^2 (p_{n0} - q_n w_0) + p_{n2} - q_n w_2,\, p_{n3} - q_n w_3,\, \cdots\right)_{ri}.\end{aligned}$$

The algorithm will converge only if the elements of P_n stay bounded for all $n \in \mathbb{N}$, which can be achieved if q_n are selected in such a way that both $p_{n1} - q_n w_1$ and $-r^2 (p_{n0} - q_n w_0) + p_{n2} - q_n w_2$ remain relatively small. It is only these two elements that may present a problem, since the rest of the sequence can be normalized in the usual way.

Remembering that

$$\frac{z}{w} = \frac{(\operatorname{Re} z)(\operatorname{Re} w) + (\operatorname{Im} z)(\operatorname{Im} w)}{(\operatorname{Re} w)^2 + (\operatorname{Im} w)^2} + i \cdot \frac{(\operatorname{Im} z)(\operatorname{Re} w) - (\operatorname{Re} z)(\operatorname{Im} w)}{(\operatorname{Re} w)^2 + (\operatorname{Im} w)^2}, \tag{75}$$

it is clear that we have to look at *two* digits at a time, and use

$$\begin{aligned}q_n &= \left\lfloor \left| \frac{p_{n0} w_0 + p_{n1} w_1 / r^2}{w_0^2 + w_1^2 / r^2} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{p_{n0} w_0 + p_{n1} w_1 / r^2}{w_0^2 + w_1^2 / r^2} \right) \\ &= \left\lfloor \left| \frac{r^2 p_{n0} w_0 + p_{n1} w_1}{r^2 w_0^2 + w_1^2} \right| \right\rfloor \cdot \operatorname{sgn}\left(\frac{r^2 p_{n0} w_0 + p_{n1} w_1}{r^2 w_0^2 + w_1^2} \right)\end{aligned}$$

as the digit selection function. This seemingly complex formula is derived directly from (75), remembering that

$$\begin{aligned}\operatorname{Re}(z_0 + z_1 (ri)^{-1} + \ldots + z_n (ri)^{-n} + \ldots) &= z_0 - z_2 r^{-2} + z_4 r^{-4} + \ldots \\ \operatorname{Im}(z_0 + z_1 (ri)^{-1} + \ldots + z_n (ri)^{-n} + \ldots) &= \frac{1}{r} \left(-z_1 + z_3 r^{-2} - z_5 r^{-4} + \ldots \right).\end{aligned}$$

3.5 Conversion to and from single-component representation

As will be shown later (Chapter 4), the single-component representation of complex numbers possesses some speed advantages in executing multiplication and division. However, in order for any number system to be of interest from the computer arithmetic standpoint, there must also be an efficient algorithmic procedure for converting numbers into the new form, as well as decoding them back into conventional form. In this section, we present the complete algorithms for the *constructor function* for creating a complex number from its real and imaginary parts, and the *selector function* for extracting real or imaginary parts of complex values. Other useful functions, such as those for replacing real or imaginary parts of a complex number while leaving the other part untouched, can be worked out in the same way.

3.5.1 The constructor function

As mentioned above, we shall assume that the real and imaginary parts of a complex number z are given by their respective radix-r signed-digit exponent representations

$$x = r^{E_x} \cdot \sum_{n=0}^{\infty} x_n r^{-n},$$
$$y = r^{E_y} \cdot \sum_{n=0}^{\infty} y_n r^{-n},$$

where $E_x, E_y \in \mathbb{Z}$ and $|x_n|, |y_n| \leq \rho_1$ ($\rho_1 \in [r/2, r-1]$) for all $n \in \mathbb{N}_0$. In order to compute a radix-ri exponent signed-digit representation of $z = x + iy$, we have to convert both x and y to radix r^2, having appropriately aligned the radix points. More precisely, we have to find an $E \in \mathbb{Z}$ and sequences $(x'_n)_{n\in\mathbb{N}_0}$ and $(y'_n)_{n\in\mathbb{N}_0}$ such that

$$x = r^{2E} \cdot \sum_{n=0}^{\infty} x'_n \left(r^2\right)^{-n},$$
$$y = r^{2E-1} \cdot \sum_{n=0}^{\infty} y'_n \left(r^2\right)^{-n},$$

which, according to (69), will then give us

$$z = x + iy = R^E \cdot (x'_0, -y'_0, -x'_1, y'_1, x'_2, -y'_2, \ldots)_{ri} \,. \tag{76}$$

The radix alignment is performed as follows:

1) if $E_x > E_y$ and $E_x \equiv 0 \bmod 2$, then we set

$$\begin{aligned} E &= E_x/2 \\ x &= r^{2E} \cdot (x_0, x_1, \cdots, x_n, \cdots)_r \\ y &= r^{2E-1} \cdot \Big(\underbrace{0, 0, \cdots, 0}_{E_x - E_y - 1 \text{ zeros}}, y_0, y_1, \cdots, y_n, \cdots \Big)_r \,, \end{aligned}$$

where $E_x - E_y - 1 = (2E - 1) - E_y \geq 0$;

2) if $E_x > E_y + 1$ and $E_x \equiv 1 \bmod 2$, then

$$\begin{aligned} E &= (E_x - 1)/2 \\ x &= r^{2E} \cdot (rx_0 + x_1, x_2, \cdots, x_n, \cdots)_r \\ y &= r^{2E-1} \cdot \Big(\underbrace{0, 0, \cdots, 0}_{E_x - E_y - 2 \text{ zeros}}, y_0, y_1, \cdots, y_n, \cdots \Big)_r \,, \end{aligned}$$

where $(2E - 1) - E_y = E_x - E_y - 2 \geq 0$;

3) if $E_x = E_y + 1$ and $E_x \equiv 1 \bmod 2$ (and therefore, $E_y \equiv 0 \bmod 2$), then

$$\begin{aligned} E &= (E_x - 1)/2 \\ x &= r^{2E} \cdot (rx_0 + x_1, x_2, \cdots, x_n, \cdots)_r \\ y &= r^{2E-1} \cdot (ry_0 + y_1, y_2, \cdots, y_n, \cdots)_r \,, \end{aligned}$$

4) if $E_x \leq E_y$ and $E_y \equiv 0 \bmod 2$, then

$$
\begin{aligned}
E &= E_y/2 \\
x &= r^{2E} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_y - E_x \text{ zeros}}, x_0, x_1, \cdots, x_n, \cdots \right)_r \\
y &= r^{2E-1} \cdot (r y_0 + y_1, y_2, \cdots, y_n, \cdots)_r ,
\end{aligned}
$$

5) if $E_x \leq E_y$ and $E_y \equiv 1 \bmod 2$, then

$$
\begin{aligned}
E &= (E_y + 1)/2 \\
x &= r^{2E} \cdot \left(\underbrace{0, 0, \cdots, 0}_{E_y - E_x + 1 \text{ zeros}}, x_0, x_1, \cdots, x_n, \cdots \right)_r \\
y &= r^{2E-1} \cdot (y_0, y_1, \cdots, y_n, \cdots)_r .
\end{aligned}
$$

All that now remains for us to do is convert the aligned sequences to radix r^2 (conversion from radix r to radix r^2 is rather straightforward), and alternate the signs of the weighted factors in accordance with (76).

Example 10. *Find a representation of $z = 1234.5 + i \cdot 9.876$ in radix $10i$. We have:*

$$
\begin{aligned}
x &= 1.2345 \cdot 10^3 = 10^3 \cdot (1, 2, 3, 4, 5)_{10} \\
y &= 9.876 \cdot 10^0 = 10^0 \cdot (9, 8, 7, 6)_{10}
\end{aligned}
$$

After aligning the decimal points (here $E_x = 3$, $E_y = 0$, therefore, $E = (E_x - 1)/2 = 1$):

$$
\begin{aligned}
x &= 10^2 \cdot (12, 3, 4, 5)_{10} \\
y &= 10^1 \cdot (0, 9, 8, 7, 6)_{10}
\end{aligned}
$$

In radix $r^2 = 100$*:*

$$\begin{aligned} x &= 10^2 \cdot (12, 34, 50)_{100} \\ y &= 10^1 \cdot (0, 98, 76)_{100} \end{aligned}$$

Finally, using (76), we find

$$z = 100^1 \cdot (12, 0, -34, 98, 50, -76)_{ri}.$$

Example 11. *Find a representation of* $z = 9.876 + i \cdot 1234.5$ *in radix* $10i$*. We have:*

$$\begin{aligned} x &= 9.876 \cdot 10^0 = 10^0 \cdot (9, 8, 7, 6)_{10} \\ y &= 1.2345 \cdot 10^3 = 10^3 \cdot (1, 2, 3, 4, 5)_{10} \end{aligned}$$

After aligning the decimal points (here $E_x = 0$*,* $E_y = 3$*, therefore,* $E = (E_y + 1)/2 = 2$*):*

$$\begin{aligned} x &= 10^4 \cdot (0, 0, 0, 0, 9, 8, 7, 6)_{10} \\ y &= 10^3 \cdot (1, 2, 3, 4, 5)_{10} \end{aligned}$$

In radix $r^2 = 100$*:*

$$\begin{aligned} x &= 10^4 \cdot (0, 0, 9, 87, 60)_{100} \\ y &= 10^3 \cdot (1, 23, 45)_{100} \end{aligned}$$

Using (76), we obtain a representation of z

$$z = 100^2 \cdot (0, -1, 0, 23, 9, -45, -87, 0, 60)_{ri}.$$

3.5.2 The selector functions

The two selector functions, Re and Im, are needed to extract the real and imaginary parts of complex values, respectively. The algorithms for the evaluation of the selector functions are essentially the inverse of those for the constructor function: given a radix-ri expansion

$$z = R^E \cdot (z_0 + z_1(ri)^{-1} + z_2(ri)^{-2} + \ldots + z_n(ri)^{-n} + \ldots),$$

the real and imaginary parts can be found as follows:

$$\operatorname{Re} z = r^{2E} \cdot \sum_{n=0}^{\infty} x_n \left(r^2\right)^{-n},$$
$$\operatorname{Im} z = r^{2E-1} \cdot \sum_{n=0}^{\infty} y_n \left(r^2\right)^{-n},$$

where

$$x_n = (-1)^n z_{2n}, \quad y_n = (-1)^{n+1} z_{2n+1}.$$

These expansions of $\operatorname{Re} z$ and $\operatorname{Im} z$ are in radix r^2; conversion from radix r^2 to radix r is trivial and was discussed in Section 2.6.1 of Chapter 2.

Chapter 4

Analysis

The object of this chapter is to give a satisfactory account of the complexity of the basic arithmetic operations on exact real numbers, and describe our implementation of the algorithms proffered in the previous chapters (also see Appendix A). We begin with a discussion of theoretical complexity by estimating the time required to compute N digits of the result of a given operation on exact numbers — or equivalently, the number of elementary operations (additions, multiplications and divisions by r) required for its completion. The space requirements, or the amount of actual physical memory needed to store the intermediate results, are more difficult to analyze because they depend on the particular implementation of the algorithms. We discuss functional and imperative language implementations, and the ways in which they can be made more efficient. Certain optimizations such as the choice of a larger radix, apply to both types of implementation, while others are specific to a particular one. We show that functional language implementations are quite wasteful of space resources, which are easier to economize in an imperative language. We conclude with a discussion of the important problem of choosing the subset of finitely represented numbers, and the choices favoured by functional and imperative language implementations.

4.1 Theoretical complexity of the algorithms

The chief and computationally most significant part of the algorithms presented in this paper is the normalization function, which is for the greater part responsible for the complexity of the four arithmetic operations.

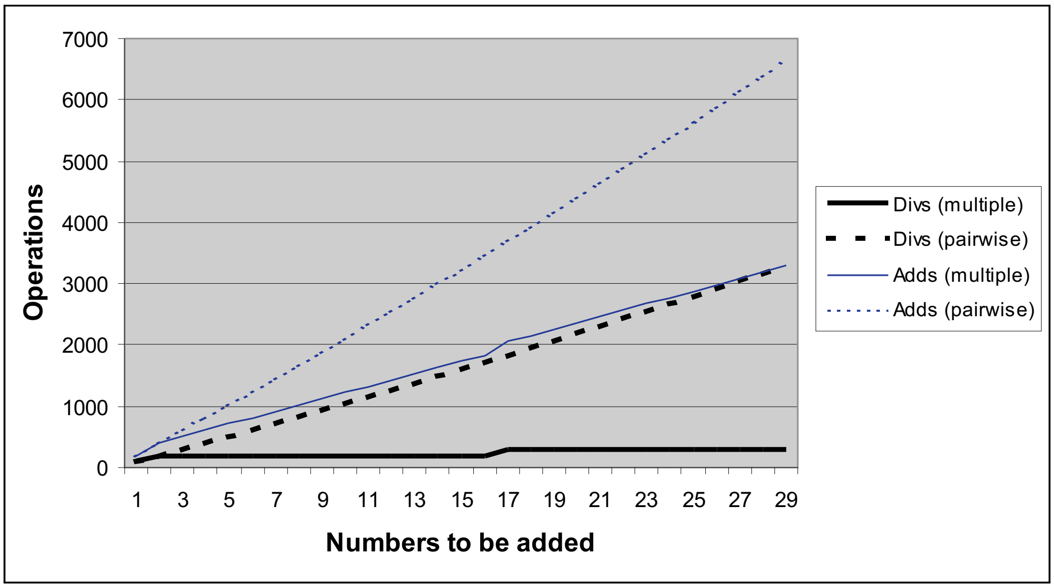
The normalization procedure relies upon *unbounded integer arithmetic* for its operation, and hence the speed of normalization is crucially dependent on the speed of same. Throughout this section, we shall be discussing algorithm complexity in terms of operations (additions, multiplications and divisions by r) on *unbounded* integers. We make this assumption mainly to ignore the possibility of overflow in integer operations; it is important to note that the numbers in our algorithms do not grow indefinitely large. We shall sometimes indicate operation counts using the notation $\alpha A + \beta M + \gamma D$, which is construed to mean "α integer additions, β integer multiplications, and γ integer divisions by r".

4.1.1 Normalization

As seen in Figure 1, normalization always requires one-digit carry-look-ahead: to produce N radix-r digits of a normalized result, it is necessary to compute $N + 1$ digits of the number being normalized, whereafter N out of the $N + 1$ digits (excluding the first one) are divided by r, and the results of the divisions — added, possibly in parallel, resulting in a total complexity of $NA + ND$ (N integer additions, and N integer divisions by r). If r is a power of 2, the divisions by r can be done by simple shifts.

Similarly, if normalization is to be performed m times, in order to obtain N digits of the result, we need $N + m$ digits of the original number, as well as $N + (N + 1) + \ldots + (N + m - 1) = m(N + (m - 1)/2)$ divisions by r and additions. The need to evaluate the extra m digits of the original number is usually referred to as the *granularity effect*.

Figure 5: Granularity in addition — the number of integer additions and divisions by r vs. the number of addends, calculated for $N = 100$ precision digits, $r = 10$ and $\rho = 6$



4.1.2 Addition and subtraction

The computation of N digits of the sum of two numbers requires $N+1$ digits of the operands, N integer divisions by r and $2N+1$ integer additions. Addition of n numbers, where $n > 2$, requires $N+m$ digits of the operands, $m(N+(m-1)/2)$ divisions by r and $m(N+(m-1)/2)+(n-1)(N+m)$ additions, where $m = \lceil \log_r (n\rho - \rho + 1) \rceil$. This is, of course, much better than the repeated binary addition $x_1 + (x_2 + (x_3 + (\cdots + x_n)))$, which results in the evaluation of $N+n$ digits of the operands, $(n-1)(N+(n-2)/2)$ divisions by r and $2(n-1)N+(n-1)^2$ additions, as shown in Figure 5.

Subtraction is only different from addition in that negation is performed beforehand. Negation, of course, does not require any look-ahead, and its complexity is simply that of reversing the sign of a number's digits.

4.1.3 Multiplication

The complexity of the multiplication algorithm depends on the value of the parameter $n \in \mathbb{N}$ that appears in (22). Let us address ourselves to the question of choosing an appropriate value for n. In principle, the algorithm will work correctly with any $n \in \mathbb{N}$, so our main concern here is to minimize the number of operations needed to compute N digits of the result. Note that when $n = 1$, the algorithm is the same as that adopted by Avizienis [5]. Choosing a larger value for n, however, helps reduce the granularity effect, as the addition of multiple numbers is significantly more efficient that the pairwise addition which occurs when $n = 1$ (see Figure 5).

Now let

$$N = pn + q, \ 0 \leq q < n.$$

We have $p + 1$ partial normalization groups (the first two groups are shown in Figure 2), each of which requires at most $m(n) = \lceil \log_r \left((n+1)\,\rho^2 - \rho + 1 \right) \rceil$ applications of `reduce` (see (22)). On account of the granularity effect, to compute N digits of the product, the normalization procedure requires m extra digits from the last partial normalization group, $2m$ extra digits from the second-last one, and so on; the first group requiring as many as pm extra digits, thus making the total number of integer divisions and additions

$$\begin{aligned}
N_{div} =& m(N + pm + (m-1)/2) + \\
& m((N-n) + (p-1)m + (m-1)/2) + \ldots + \\
& m((N-pn) + (m-1)/2) \\
=& m\left(N + (N-n) + (N-2n) + \ldots + (N-pn)\right) + \\
& m^2(p + (p-1) + \ldots + 1) + m(m-1)(p+1)/2 \\
=& \frac{1}{2} m\,(p+1)\,(N + q + mp + m - 1)\,.
\end{aligned}$$

The corresponding formula for the number of integer multiplications of the operands' digits is

$$\begin{aligned} N_{mult} &= \frac{N(N+1)}{2} + (q-1)m + n\cdot(2m + \ldots + (p+1)m) \\ &= \frac{N(N+1)}{2} + m\left(\frac{p^2+3p}{2}n + q - 1\right) \end{aligned}$$

Thus, we have to choose n such as to minimize the two functions

$$\begin{aligned} N_{div}(n,N) &= \frac{1}{2}m(n)\cdot(p+1)\left(N + q + m(n)\cdot p + m(n) - 1\right), \\ N_{mult}(n,N) &= \frac{1}{2}N(N+1) + m(n)\left(\frac{p^2+3p}{2}n + q - 1\right), \end{aligned}$$

where

$$\begin{aligned} p &= \left\lfloor \frac{N}{n} \right\rfloor, \quad q = N \mod n, \\ m(n) &= \left\lceil \log_r \left((n+1)\,\rho^2 - \rho + 1\right)\right\rceil. \end{aligned}$$

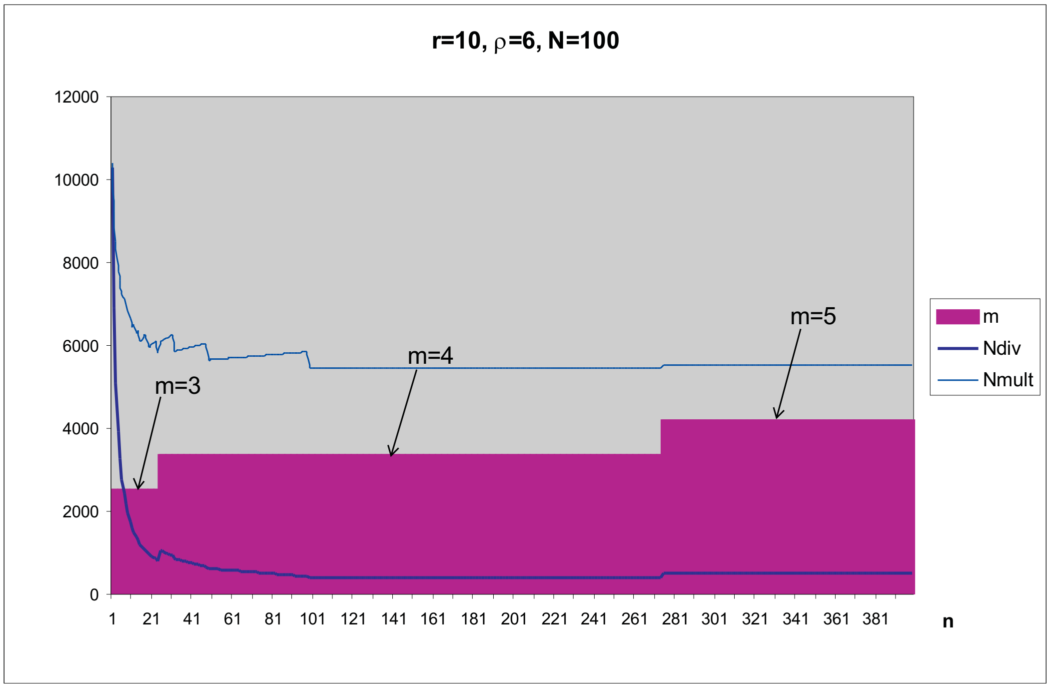
For instance, let $r = 10$, $\rho = 6$, then

$$m(n) = \begin{cases} 2, & \text{if} \quad n = 1 \\ 3, & \text{if} \quad 2 \le n \le 26 \\ 4, & \text{if} \quad 27 \le n \le 276 \\ 5, & \text{if} \quad 277 \le n \le 2776 \\ \ldots & \ldots \quad \ldots \end{cases}$$

and the corresponding minimal values of N_{div} for $N = 100$ are

$$\begin{aligned} N_{div}(1,100) &= \frac{1}{2}\cdot 2\cdot 101\cdot 301 = 30401 \\ N_{div}(26,100) &= \frac{1}{2}\cdot 3\cdot 4\cdot 133 = 798 \\ N_{div}(276,100) &= \frac{1}{2}\cdot 4\cdot 1\cdot 203 = 406 \\ N_{div}(2776,100) &= \frac{1}{2}\cdot 5\cdot 1\cdot 204 = 510 \end{aligned}$$

Figure 6: Choice of n for multiplication — the number of integer multiplications N_{mult} and divisions N_{div} vs. n calculated for $N = 100$ precision digits



These data have been summarized in graphical form in Fig. 6 as plots of N_{div} and N_{mult} versus n for $N = 100$. It is apparent that $N_{div}(n)$ and $N_{mult}(n)$ behave similarly for other values of N, r and ρ. One can see that the optimal value of N_{div} is attained when $n = N + 1$, in which case $p = 0$, $q = N$, and the total number of divisions is $N_{div}(N + 1, N) = mN + m(m - 1)/2$. Since the number N of required precision digits is generally unknown in advance, it is reasonable to choose some *fixed* value of n that would ensure reasonable performance of the algorithm for all N. It is also clear that we may only choose n out of

$$n_k = \max \{ n \in \mathbb{N} \mid m(n) \leq k \}, \ k \in \mathbb{N},$$

because if $n' > n''$ and $m(n') = m(n'')$, we have $N_{div}(n') \leq N_{div}(n'')$.

In principle, the larger the value of n, the better; except when n is vastly larger than N, the number of operations N_{op} will continue to grow with N (due to the increasing of m). On the other hand, choosing a large value of n would imply large values of the sequence entries (up to $(n+1)\,\rho^2$ — see (22)) which, if exceeded the threshold for representing integers (usually the size of the machine word), would result in slower integer operations. The values of n corresponding to $m = 2$ are obviously inadequate, resulting in an unnecessarily large number of operations (e.g. Avizienis's algorithm), but any of the numbers $n_3, n_4, \ldots$ appear to be equally suitable for the value of n (in terms of operation counts). Of course, the larger the m in n_m, the more normalizations (and therefore more space to hold the intermediate results) will be required; hence we have used $n = n_3$ in our implementation (e.g. for $r = 10$ and $\rho = 6$, $n = n_3 = 26$).

4.1.4 Division

Division can be analyzed in much the same way as multiplication, and is also quadratic. For simplicity's sake, we shall assume that the partial remainders are normalized *fully* — as remarked above, the actual time estimates will only be better.

By (26), to determine the N-th digit q_N of the quotient, the division algorithm must compute the first digit of the N-th partial remainder P_N, which according to (25), involves evaluation of P_{N-1} and D to 3 digits (an extra digit is required because of the multiplication by r), that in turn demands P_{N-2} and D to 5 digits, and the domino effect applies to the rest of the partial remainders, so that P_0 will be evaluated to $2N+1$ digits. In summary, we will have $N^2 = 1+3+\ldots+2N-1$ integer divisions by r and additions from the normalization of P_N, P_{N-1}, $\cdots$, P_1, plus N more divisions (not necessarily by r) from the digit selection guesswork in (26), as well as $N(N+2) = 3+5+\ldots+(2N+1)$ additions and multiplications of the quotient digits q_n by the digits of the divisor D in (25).

4.1.5 Complex arithmetic

Let us now compare the theoretical complexity of exact complex arithmetic based on the radix ri representation described in Chapter 3, and the conventional pairs-of-reals approach using exact reals represented in radix r^2 (as per Chapter 2). We are primarily interested in the number of multiplications and divisions involved, since additions are subtractions are relatively inexpensive.

Let $z = z_1 + iz_2$ and $w = w_1 + iw_2$ be two complex numbers, and suppose we want to evaluate the real and imaginary parts of $z \diamond w$ to N radix-r^2 digits, where $\diamond$ is one of the operations $\{+, -, \times, \div\}$. It is clear that N most significant radix-r^2 digits of the real and imaginary parts of $z \diamond w$ will be encoded in $2N$ most significant radix-ri digits of $z \diamond w$.

Addition or subtraction of two complex numbers involves two real additions or subtractions, and because we need to evaluate twice as many digits in radix-ri, the complexity of the addition/subtraction operation is the same as that in radix r^2.

The complexity of multiplication was analyzed in Section 4.1.3, and described by the following formulae

$$\begin{aligned} N_{mult}(N) &= \frac{1}{2}N(N+1) + m\left(\frac{p^2+3p}{2}n + q - 1\right), \\ N_{div}(N) &= \frac{1}{2}m \cdot (p+1)\,(N + q + m \cdot p + m - 1)\,, \end{aligned}$$

where for radix r^2 we have

$$m = 3, \quad p = \left\lfloor \frac{N}{n} \right\rfloor, \quad q = N \bmod n, \quad n = \frac{r^{2m} + \rho - 1}{\rho^2} - 1.$$

When complex numbers are represented by pairs of reals, one complex multiplication requires four real multiplications, hence we have

$$\begin{aligned} N_{mult}^{(radix\ r)}(N) &= 2N(N+1) + 2m\left(p^2 n + 3pn + 2q - 2\right), \\ N_{div}^{(radix\ r)}(N) &= 2m \cdot (p+1)\,(N + q + m \cdot p + m - 1)\,. \end{aligned} \tag{77}$$

In radix-ri, however, we have

$$\begin{aligned} N_{mult}(N') &= \frac{1}{2}N'(N'+1) + m\left(\frac{(p')^2+3p'}{2}n + q' - 1\right), \\ N_{div}(N') &= \frac{1}{2}m\cdot(p'+1)\left(N' + q' + m\cdot p' + m - 1\right), \end{aligned} \tag{78}$$

where

$$N' = 2N, \quad m = 3, \quad p' = \left\lfloor \frac{N'}{n} \right\rfloor, \quad q' = N' \bmod n, \quad n = \frac{r^{2m}+\rho-1}{\rho^2} - 1. \tag{79}$$

Substituting (79) into (78), we obtain

$$\begin{aligned} N_{mult}^{(radix\ ri)}(N) &= N(2N+1) + m\left(\frac{p^2+3p}{2}n + q - 1\right), \\ N_{div}^{(radix\ ri)}(N) &= \frac{1}{2}m\cdot(p+1)\left(2N + q + m\cdot p + m - 1\right), \end{aligned} \tag{80}$$

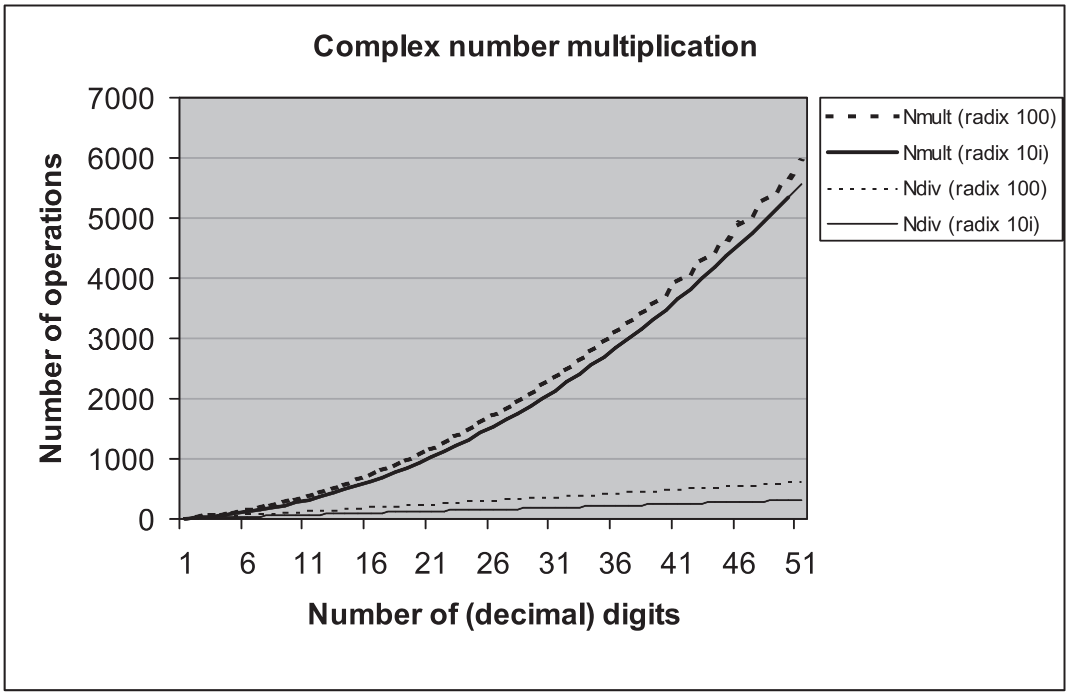
where

$$m = 3, \quad p = \left\lfloor \frac{2N}{n} \right\rfloor, \quad q = (2N) \bmod n, \quad n = \frac{r^{2m}+\rho-1}{\rho^2} - 1.$$

Comparing (77) and (80), one can see that (80) is asymptotically better than (77), which is demonstrated in Figure 7 for $r = 10$ and $\rho = 51$ ($n = 228$).

Division of complex numbers represented as pairs of reals involves *six* real multiplications and *two* real divisions, and therefore is *significantly* more efficient in radix-ri format: the radix-ri division algorithm is virtually the same as that in radix r^2 (with the exception of the quotient digit selection function which has a more complicated form in radix ri), hence its complexity is comparable with that of the radix-r^2 algorithm. Although division is quadratic (and $2\cdot f(N)$ is always better than $f(2N)$ for a quadratic complexity function $f(N)$), the radix-ri method is asymptotically faster because it avoids the overhead of the six real multiplications (also see Table 3).

Figure 7: Complexity of multiplication in radices $10i$ and 10.



4.2 Design and implementation

In this section we discuss the design of an infinite precision arithmetic package based on the redundant signed-digit representation of exact numbers, and the data types required to implement the algorithms in functional and imperative languages. Although imperative implementations are significantly faster and more memory-conscious, functional programs are an order of magnitude shorter and therefore easier to develop, modify and test. The functional programmer need not be concerned with issues such as memory management or garbage collection, and can concentrate his efforts on the more important tasks. A working functional implementation is also useful as a prototype of an imperative one, and therefore is more reasonable to enter upon.

4.2.1 Implementation in a functional language

A program written in a functional language consists entirely of functions. The program itself is a function which receives the input as its argument and returns the output as its result. The main function is defined in terms of other functions, which in turn are defined in terms of more functions, and so forth — until the bottom-layer functions are defined in terms of language primitives.

An important feature of modern functional programming languages, such as Miranda or Haskell, is that they incorporate *lazy evaluation*. This means that arguments are passed to functions unevaluated, and are only evaluated when (and if) they are needed. Furthermore, if an argument is a composite object, such as a list, it is only evaluated to the extent required by the function applied to it. It is precisely this facility that permits us to define and reason about infinite objects: for example, the following recursive definition defines the infinite list of zeros:

```
zeros = 0 : zeros
```

The Miranda language

The chosen implementation language was the functional programming language Miranda [85]. Miranda was chosen for several reasons:

- it incorporates lazy evaluation and infinite lists, which makes it possible to write down definitions of infinite data structures,
- it automatically performs garbage collection,
- it has built-in arbitrary length integer arithmetic, and
- Miranda permits the definition of "abstract types", whose implementation is hidden from the rest of the program.

The latter feature is a flexible mechanism for modular design, which allows the Miranda programmer to introduce an "exact real" or ("exact complex") abstract type and export the names of related functions.

Data types

Lazy evaluation creates the possibility of defining and actually computing with exact real numbers. An exact real number is essentially defined by its exponent and a lazy list of its digits (assuming that the radix r is fixed). The corresponding Miranda data type is

```
real == (num,[num])
```

Here `num` is the number type, and `[num]` is the type of lists whose elements are of type `num`. To form lists, we use Miranda's lazy list constructor `(*:*)`, which is lazy in its both arguments, although we do not exploit the laziness in the first argument in our algorithms. A list of type `[num]` might be finite or infinite, and a function that takes such a list for an argument has no way of determining its (in)finiteness. Therefore, we can either promote all finite lists to infinite by appending an infinite list of zeros at the end of their mantissas, or include an extra parameter of type `bool` in the `real` data type to distinguish between finite and infinite lists:

```
real == (bool,num,[num])
finite = True
infinite = False
```

Although finite and infinite numbers have quite different logical properties and do not belong in the same data type (see Section 4.2.5), lists of type `finite` can be viewed as a shorthand for infinite lists with zeros at the end, but without the additional overhead of infinite list expansions. For instance, if we wish to compute a simple expression such as $(1 + 2)$, we need never perform more than a finite amount of computation. However, if both 1 and 2 were represented by infinite streams of digits, the computation would proceed indefinitely. The type function

```
typ (t,e,m) = t
```

can be used to check whether the mantissa of a given number is finite or infinite, and the functions

```
expo (t,e,m) = e
man (t,e,m) = m
```

can be used to extract its exponent and mantissa respectively. Notice that `infinite` is the weaker type: if an expression involves both finite and infinite numbers, the result is usually infinite unless the infinite numbers are unused:

```
|| weaker_type x y is infinite if either x or y is infinite
weaker_type x y = typ x & typ y
```

Functions on exact real numbers

Almost every function that operates on exact reals has a counterpart that operates on the mantissas of its arguments. A good example is the addition function, which first adjusts the mantissa of one of the arguments to align the radix points, and then adds the mantissas themselves. The following Miranda function carries out the latter addition:

```
add (a:x) (b:y) = (a+b) : add x y
add x [] = x
add [] y = y
```

(The last two lines account for the case when at least one of the operands is finite.)

Of course, the function `add` produces an *unnormalized* result even if given two normalized lazy lists. To normalize the result we use the function `reduce`, described in Section 2.3:

```
reduce [] = []
reduce (a:x)
  = add (a:map fst divrem) (map snd divrem)
    where
```

```
divrem = map f x
|| here r = 10, rho = 6
f n = (m,d),          if n>=0 & m<6
    = (m-10,d+1),     if n>=0 & m>=6
    = neg2 (f (-n)), otherwise
      where
      d = n div 10
      m = n mod 10
      neg2 (x,y) = (-x,-y)
```

Using `reduce`, we can now define the complete addition function `plus` as follows:

```
plus::real->real->real
plus a b
  = (t,e,reduce m), if expo a >= expo b
  = plus b a,       otherwise
    where
    t = weaker_type a b
    e = expo a
    m = add (man a) (man b),                 if expo a = expo b
      = add (man a) (zeros (expo a-expo b) ++ man b), otherwise
    zeros n = rep n 0
```

Similarly, subtraction can be performed as

```
minus x (t,e,m) = plus x (t,e,map neg m)
```

where `neg` is the negation function, and `map` is the built-in Miranda function, which being applied to a function and a list returns a copy of the list in which the given function has been applied to every element.

The complete Miranda source code listings for other functions on exact reals described in the previous chapters can be found in Appendix A.

4.2.2 Implementation in a compiled imperative language

In this section we shall discuss the data types and structures needed to implement the algorithms in the programming language C. The choice of the C language to be used for this purpose is governed by several considerations: a) it is operator-rich and has a reasonable variety of built-in data types, b) it gives the programmer direct access to most capabilities of the hardware, and c) it has been ported to virtually all existing platforms, being the native language of the UNIX operating system. The C++ language is also well suited for the task; in fact, C++ has several advantages over C:

- an abstract *class* of exact real or complex numbers can be defined in C++ with a set of public operations,
- *constructors* and *destructors* can be used to allocate and release resources associated with exact numbers,
- C++ allows *operator overloading*, which lets users manipulate exact numbers in the usual way.

Representation of lazy lists in C

In functional languages, lists are defined by

```
listof X ::= nil | cons X (listof X)
```

which means that a list of `X`'s (whatever `X` is) is either `nil`, representing an empty list, or it is a `cons` of an `X` and another list of `X`'s. A `cons` is by definition a list whose first element is the `X` and whose subsequent elements are the elements of the other list of `X`'s (here `X` may stand for any data type). For example,

```
[]         means    nil
[1]        means    cons 1 nil
[1,2,3]    means    cons 1 (cons 2 (cons 3 nil))
```

It seems natural to represent lazy lists of digits in C by linked lists of *nodes*, which can be either *cons* (or *list*) *nodes* (`tlist`) which hold the values of the elements evaluated so far, or *application nodes* (`tapp`) which hold the pointers to a continuation function and its argument(s):

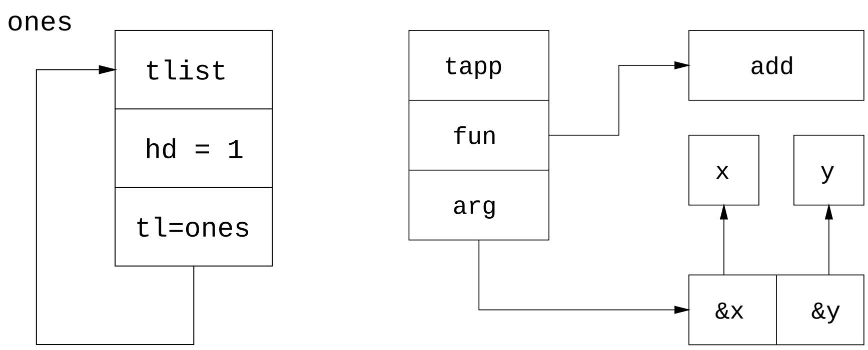
```
struct rnode {   /* real number node */
  enum  { tlist, tapp } tag;
  union {
     struct {    /* tlist */
       int           Rhd;    /* head of the list */
       struct rnode *Rtl;    /* tail of the list */
     } ulist;

     struct {    /* tapp */
       void (*Rfun) ();      /* function to apply */
       struct rnode **Rarg;  /* pointer to a (list of) argument(s) */
     } uapp;
  } Ru;
};

/*  some useful macro definitions */
#define hd Ru.ulist.Rhd  /* head of a list */
#define tl Ru.ulist.Rtl  /* tail of a list */

#define fun Ru.uapp.Rfun /* application of a function */
#define arg Ru.uapp.Rarg /* its argument(s) */
```

The function to apply in an application node must be a function defined on lazy lists, such as `add` or `mult`, designed to always return a lazy list whose first element is a *list* node. A lazy list of digits typically has multiple list nodes and a single application node, although neither is strictly required. For example, the



(a) A single tlist node representing an infinite lazy list

(b) A single application node representing the sum of two numbers x and y

Figure 8: Examples of tlist and tapp rnodes

following infinite repeating list of digits

```
repeat 1 = [1,1,1,1,1,1,...]
```

may be represented by a single list node (Figure 8(a)), and the sum of two numbers `x` and `y` is (initially) represented by just one application node shown in Figure 8(b).

The following evaluation function, `eval`, can be used to expand (evaluate more elements of) a lazy list:

```
void eval (struct rnode *x)
{
  /* if x is a list, we leave it alone;
     otherwise, we evaluate one term    */

  if (x->tag == tapp)
     (*x->fun) (x, x->arg);
}
```

An application of `eval` is tantamount to evaluation on demand: if the first node of a list is an application node, `eval` applies the continuation function to its argument(s); otherwise (the list node has already been evaluated), it does nothing. Note that the arguments supplied to the continuation function might also have to be `eval()`uated, and the information on how far to expand the arguments must be encoded in the function applied to them.

Assuming that finite lists are terminated by a `NULL` pointer, the following function uses `eval` to print the elements of a lazy list (it will never terminate if the argument is an infinite list):

```
void print (struct rnode *x)
{
    for (; x->tl != NULL ; x = x->tl) {
       eval(x);
       printf(''%d, '', x->hd);
    }
}
```

Representation of exact reals

An exact real number is a dynamic object, much like a pointer, only with a significantly more complex set of dependencies. When an exact real number is first declared, a fixed amount of memory is reserved to hold the pointer to a structure representing the number. When the number is initialized (assigned a value), more memory is allocated to hold the contents of the structure itself; the memory can dynamically expand when more precision is required. An exact real number cannot be deleted until the last reference to it disappears: for example, if x and y are exact real numbers, neither can be deleted as long as their sum $x + y$ is in use. To ensure that this is the case and to enable garbage collection, a reference count must be included with each real number to keep track of its number of references. A real number is considered reclaimable whenever its reference count becomes zero. The runtime system (or potentially the compiler) must be responsible for adjusting the reference count every time a number is updated or copied. Whenever an exact real number is assigned to, e.g. $x = y$, the referent of y has its count incremented, and the former referent of x has its count decremented.

For instance, the following assignment:

```
x = add(x,y);
```

where `x` and `y` are exact real numbers, will be correctly evaluated only if the location of the original structure representing `x` is saved in a temporary variable (`tmp`), as shown in Figures 9 and 10.

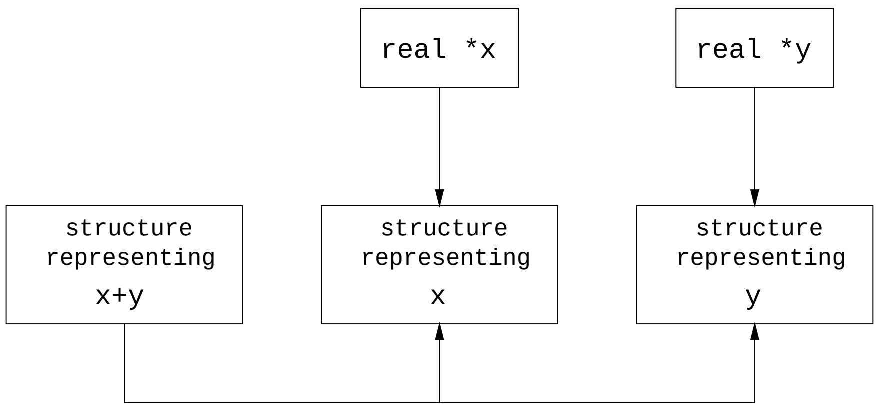


Figure 9: Dependencies of `add(x,y)`, where `x` and `y` are exact real numbers

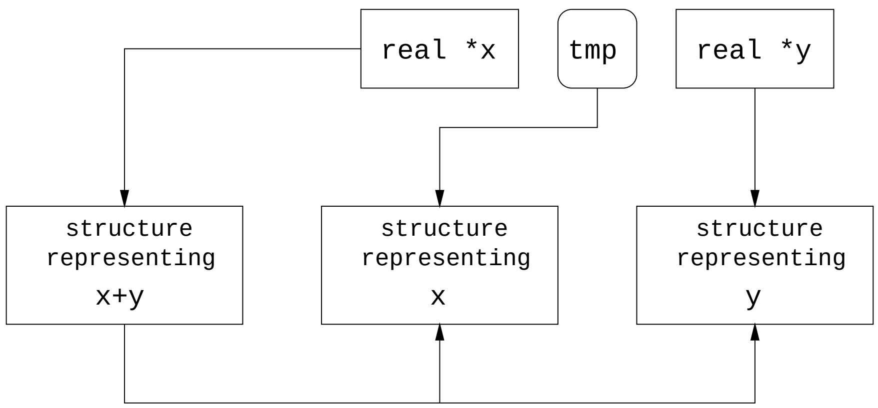


Figure 10: The appearance of a temporary object in the evaluation of `x = add(x,y)`

If `x=x+y` is later deleted, the memory occupied by `tmp` (original `x`) must be deallocated, which is only possible if the location pointed at by `tmp` is referenced by the structure representing `x+y`. Note that this does not obviate the need for reference counters, as `tmp` might be referenced by another number and therefore cannot be deleted until the last reference to it disappears, as in the following example:

```
x = exact(2.0); // ref(x)=0, ref(exact(2.0))=1
y = exact(3.0); // ref(y)=0, ref(exact(3.0))=1
z = sqroot(x);  // x is referenced by z, refcount incremented:
                // ref(x)=1
x = add(x,y);   // old x becomes a temporary object (tmp) and is
                // referenced by add(x,y)
                // ref(tmp)=2, ref(y)=1, ref(x)=0
delete(x);      // x is deleted, dependencies refcounts decremented:
                // ref(tmp)=1, ref(y)=0
delete(y);      // y is deleted, storage deallocated since ref(y)=0
                // tmp still exists, because ref(tmp)=1
delete(z);      // ref(tmp)=0, temporary object tmp can be deleted
```

This example shows that a sequence of computations involving exact real numbers can generate *temporary objects* unbeknownst to the programmer or user. Each of this objects has a *reference counter* and must be automatically destroyed when the last number referencing the object is deleted. The reference counter is potentially an *unbounded* integer, although on current architectures memory considerations would most likely preclude the referencing of one object by more than `maxint` (2^{32} or 2^{64}) numbers, and therefore we can safely declare it to be an `int` (one possible course of action if an object is referenced by more than `maxint` numbers is to mark it as permanent). Any object may of course be itself dependent on other temporary (or non-temporary) objects, and once deleted, the reference counts of all the dependencies must be decremented. Reference counting in general, and as a garbage collection technique in particular, remains an active and much studied area of research (for more details, the reader can refer to e.g. [38]).

Taking the aforesaid into account, the following structure can be used to represent exact reals:

```
struct real
{
    enum { inf, fin, rep } type; /*     type     */
    int expo;                    /*   exponent   */
    struct rnode *man;           /*   mantissa   */

    int ref;                     /*  ref count   */
    struct real **deps;          /* dependencies */
};
```

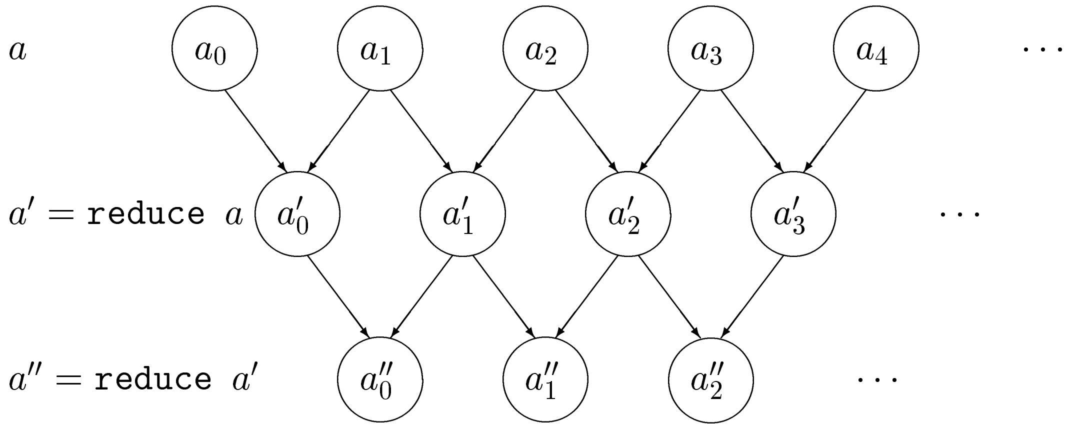
The `type` field is optional and has been introduced into this structure to avail ourselves of a finitely represented data type, as discussed below in Section 4.2.5. The exponent is generally an unbounded integer; an `int` is used here for the sake of simplicity.

Implementation of reduce

A program written in a functional language contains no assignment statements, and variables, once given a value, can never change. This fundamental property enjoyed by functional languages is known as *referential transparency*, and usually considered to be one of the main advantages of functional programming, because it eliminates a major source of programming errors caused by the side-effects of the order of evaluation, which sometimes results in the accidental modification of variable values. In a referentially transparent program, the order of execution is irrelevant, because a function call can have no side-effect other than to compute its result, and one can freely replace variables with their values and vice versa.

Consequently, a function cannot modify its argument(s), and in particular, the `reduce` function, applied to a list of numbers, always returns a new list. If a list of digits needs to be normalized several times, each instance of `reduce` generates another copy of the list. Hence, for each application of `reduce` we have no choice but to keep copies of all previous unnormalized lists (see Figure 11).

Figure 11: Multiple applications of `reduce`: functional language implementation



This approach is very wasteful of space, and can be easily improved in an imperative language. Indeed, an imperative version of `reduce` could be designed to modify the *existing* list instead of creating a new one. Since all functions favour normalized lists over unnormalized ones, *replacing* a list with its normalized copy does not normally create any problems. We now proceed to show how this could be coded in an imperative language.

Let `(a:b:x)` be the list we would like to normalize (Figure 12). We assume that it begins with a list node; if it does not, we can always convert the application node into a list node by calling `eval`.

One application of `reduce` changes the list to the one shown in Figure 13, which is initially unevaluated (it starts with an application node). After applying `eval`, we obtain the node shown in Figure 14, where the functions f_1 and f_2 are

Figure 12: A `tlist` rnode representing `(a:b:x)`

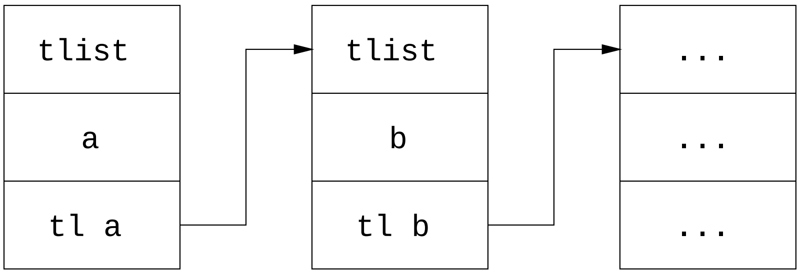
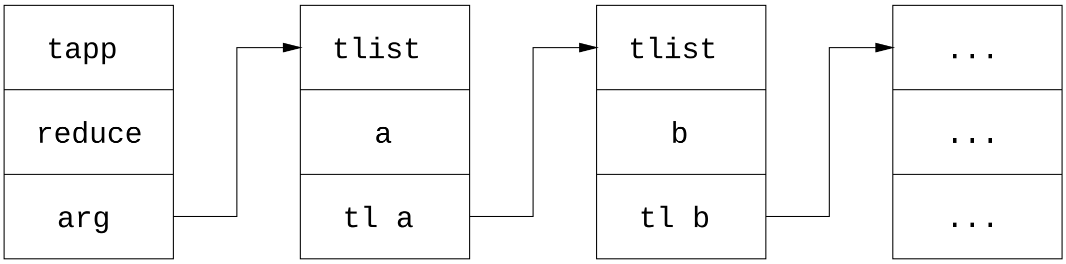


Figure 13: A `tapp` rnode representing `reduce(a:b:x)`



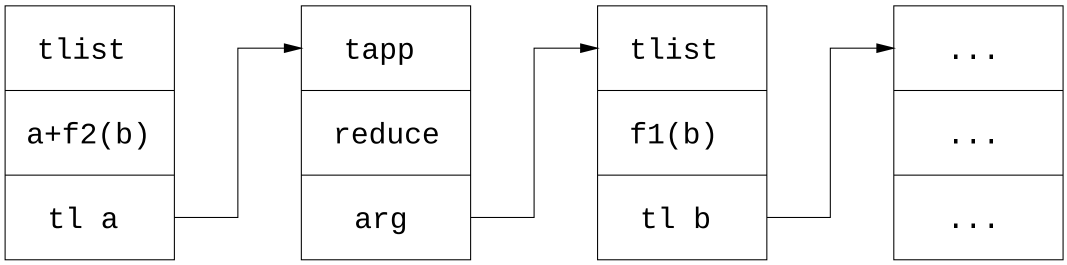
obtained from (15) and defined as follows:

$$f_1(x) = \begin{cases} -f_1(-x), & \text{if} \quad x < 0 \\ x \bmod r, & \text{if} \quad x \bmod r < \rho \\ (x \bmod r) - r & \text{if} \quad x \bmod r \geq \rho \end{cases}$$

$$f_2(x) = \begin{cases} -f_2(-x), & \text{if} \quad x < 0 \\ x \operatorname{div} r, & \text{if} \quad x \bmod r < \rho \\ (x \operatorname{div} r) + 1 & \text{if} \quad x \bmod r \geq \rho \end{cases}$$

If we want to normalize the list further, we can repeat the steps shown in Figures 12, 13, and 14 as many times as needed. Each call to `reduce` prepends the list with an application node containing a pointer to `reduce`, which is replaced with a list node upon a call to `eval` according to Figure 14. Note that each application of `reduce` adds an extra term to the list being normalized, which of course is the direct consequence of the granularity effect described in Section 4.1.1.

Figure 14: A `tlist` rnode representing `eval(reduce(a:b:x))`



4.2.3 Efficiency

In this section we show some timing results that illustrate the effect of using exact real and complex arithmetic. All our numerical experiments with the functional implementation were carried out in Miranda (version 2.018), and for our imperative implementation we used the EGCS C compiler (version 2.91.66). The results quoted in this section were obtained on an AMD K6-2 (333 MHz) workstation running the FreeBSD 4 operating system, and also reproduced (with slightly different timing results) on a Sun UltraSPARC station running Solaris 7 using the same release of Miranda and Sun's C compiler. The figures in Tables 1, 2 and 3 are the AMD figures for a (10,6) representation of exact reals implemented in Miranda.

Shown in Table 1 are the timing comparisons at 100, 500 and 1000 significant decimal digits for some operations on real numbers. The real numbers used in the experiments were random, and the results are average times based on many calls to each function; hence all timings are approximate. The averages appeared to be consistent only for the four basic arithmetic operations and square root. The time required for the evaluation of transcendental functions varied greatly from one experiment to another, depending on the arguments supplied. This can be attributed to the nature of the evaluation algorithms which depend on power series expansions whose rate of convergence can vary significantly. The results for the transcendental functions are therefore only provided for several fixed arguments in order to give a rough idea of the timing.

Table 1: Real arithmetic timing: elementary operations (seconds)

Precision (decimal digits)	100	500	1000
Addition	0.01	0.05	0.1
Subtraction	0.01	0.05	0.1
Division	0.5	3	12
Multiplication	0.3	3	16
Square root	0.5	12	72
e	1	11	43
$\sin 1$ (and $\cos 1$)	5	165	(N/A)
$\log 1.5$	7	485	(N/A)

Table 2 demonstrates how a chain of dependencies can adversely affect the performance of exact real arithmetic when we successively evaluate

$$a_1 = \sqrt{2}, \quad a_2 = \sqrt{\sqrt{2}}, \quad a_3 = \sqrt{\sqrt{\sqrt{2}}}, \quad \cdots$$

at 100 decimal digits precision.

Table 2: Real arithmetic timing: complex expressions (seconds)

Number (100 decimal digits)	Time (seconds)
$a_1 = \sqrt{2}$	0.83
$a_2 = \sqrt{a_1}$	1.58
$a_3 = \sqrt{a_2}$	3.25
$a_4 = \sqrt{a_3}$	6.08
$a_5 = \sqrt{a_4}$	10.58
$a_6 = \sqrt{a_5}$	19.50
$a_7 = \sqrt{a_6}$	34.10
$a_8 = \sqrt{a_7}$	68.05
$a_9 = \sqrt{a_8}$	131.59
$a_{10} = \sqrt{a_9}$	285.67

Note that the difference between the evaluation times of a_{n+1} and a_n almost doubles with each iteration. This ill behaviour can be attributed to the excessive copying of lists described on p. 128, which to some extent is unavoidable in functional languages.

Table 3 compares the timings for various functions on exact complex numbers implemented as pairs of exact radix-10 reals and in radix $10i$. The radix $10i$

timings include the conversion to and from radix 10 (hence the actual timings are even better).

Table 3: Complex arithmetic timing: radix $10i$ and 10 implementations

Operation	Precision (decimal digits)	Radix 10 (sec)	Radix $10i$ (sec)
Multiplication	50	1.10	0.58
Multiplication	100	2.37	1.25
Multiplication	200	6.13	2.42
Division	50	4.75	3.00
Division	100	14.00	6.65
Division	200	44.50	17.95

Finally, Table 4 shows the timing comparisons of the Miranda and C implementations of David Turner's e digits algorithm at different precision (the source code is included in Appendix C).

Table 4: Evaluation of e: Miranda and C implementations

Precision (decimal digits)	Miranda (seconds)	C (seconds)
100	0.93	0.07
200	2.42	0.27
300	4.65	0.50
400	8.15	0.85
500	11.08	1.23
600	16.65	1.77
700	21.42	2.32
800	27.25	2.99
900	34.13	3.69
1000	43.13	4.43

Conclusions:

- single operations on exact numbers are reasonably efficient, but the evaluation of complex expressions leads to a serious performance degradation,
- complex arithmetic operations are more efficient in an imaginary radix system, and
- our "naïve" imperative implementation is approximately 10 times faster than the functional one.

In the following section we shall discuss the shortcomings of the existing implementations and the ways in which they can be made more efficient.

4.2.4 Optimization

We can identify a number of ways in which the implementations can be improved, the most important ones being the following:

- The choice of radix. Since real numbers are represented on a computer by linked lists of digits, choosing a larger radix can help reduce the housekeeping overheads. All hardware operations on integers that can be represented in one word take the same amount of time regardless of their size, therefore a larger radix should improve performance (as long as the integers remain within the size of the machine word).

- The use of fixed and arbitrary precision integers. In many functions unbounded integers can be replaced by their fixed-size counterparts. For example, it is unlikely that a number would have more than 2^{32} dependencies, or that the user would ever request more than 2^{32} digits of a number. Since the evaluation of most functions on exact numbers involves a great many integer operations, avoiding arbitrary length integers whenever possible can lead to a significant efficiency gain.

- Space-time optimization. Space and time optimizations are often mutually exclusive — it is possible to optimize the time at the expense of the space, and vice versa. For example, each step (25) in the division algorithm requires evaluation of a divisor multiple $q_n D$. The guessed digits q_n are between $-\rho$ and ρ, and when the radix r is a small number (e.g. 10), we can avoid recalculation of $q_n D$ by keeping all of the 2ρ divisor multiples in memory, and re-use them every time the same multiple is needed.

- Data re-use. It is often possible to re-use intermediate data produced in seemingly unrelated computations. For example, elementary functions use power series expansions, which require the evaluation of the powers x^n for a given x. If several elementary functions of x (or x^n) are to be evaluated, it seems natural to re-use the powers of x than had been previously computed. In order to do so, the structure that represents x must reference a list of pointers to the powers of x. Generally, the structure representing an exact number might reference a number of commonly used dependencies to allow all functions of the same argument to share as much intermediate data as possible. The decision as to whether to include a reference to a function foo of x in the structure representing x must be made on a function-by-function basis and is also a case of space-time optimization — including more references in the structure increases its size and does not always improve performance (for instance, if an x is only used once, the extra references are a waste of space).

- Parallelization. Many functions, such as `reduce`, can be defined in terms of operations that can be executed in parallel. Such functions could be multi-threaded, although some redesigning might be required to ensure that no deadlocks could occur, and also to limit the number of threads that can be spawned at any given time (if any call to `reduce` were allowed to create multiple threads, we would very soon reach the thread limit imposed by the operating system).

It is clear that some operations, particularly the elementary functions, will always be expensive regardless of the optimizations, unless other evaluation methods are devised. Implementations in an imperative language, such as C or C++, are more optimizable than functional implementations, because the programmer has full control over the data structures and the features provided by the operating system (e.g. threads), while using a functional language means relying on its handling of functions, numbers and lists, and multi-threading is also problematic.

4.2.5 Choosing the set of finitely represented numbers

Exact real and complex numbers are defined as infinite objects, be it limits of Cauchy sequences, infinite expansions or continued fractions. Arithmetically, 1, 1.0 and $1.0000\cdots$ (infinite number of zeros after the decimal point) are all equal to one, and in principle we can view *all* real and complex numbers (including the integers) as being infinite. From the computing standpoint, however, the logical properties of finite and infinite numbers are quite different; for example, equality is decidable on the former but not on the latter. Therefore, we must also provide a data type in which numbers are represented *finitely*, since using an infinite data type precludes effective computation of many programming tasks.

The choice of a subset of finitely representable numbers is an important design decision, and generally, the larger the class of numbers with finite representations, the better. For instance, all integers must be finitely represented, and ideally the set of finitely represented numbers should be closed under as many arithmetic operations as possible. The smallest subset of the real numbers containing the integers and closed under the basic arithmetic operations $\{+, -, *, /\}$ is known to be the field of rational numbers [47]. Although the rational numbers may seem to be a very good candidate for the class of finitely representable numbers, there are a number of pragmatic reasons for not choosing the rationals [62].

In essence, the rational numbers can be represented in radix-r by infinite recurring sequences of digits. Although the representation of rationals as repeating radix-r numbers can be made consistent with the previously described redundant radix-r representations of the reals, and arithmetic operations — modified to perform lazily on the rationals, any efficient implementation based on this representation must overcome nontrivial technical challenges, such as being able to recognize a state of computation that has occurred before, or having to deal with representations of very great lengths. It is also interesting to note that when rationals are represented in radix-r by sequences of *signed digits*, they may no longer recur, as demonstrated by the following example.

Let

$$x = \frac{5}{9} = 0.555555\ldots = (-1, [5, 5, 5, 5, 5, 5, \cdots])_{10},$$

and define the sequence $(a_n)_{n\in\mathbb{N}} = (1, 3, 6, 10, 15, \ldots)$ by the following recurrence relation

$$a_n = a_{n-1} + n, \quad a_1 = 1.$$

Applying $f_{(a_n)}$ (Definition 9) to the mantissa of x, we obtain the sequence

$$[6, \overline{5}, 6, \overline{5}, 5, 6, \overline{5}, 5, 5, 6, \overline{5}, 5, 5, 5, 6, \overline{5}, 5, \ldots],$$

which is also a valid representation of x in radix 10:

$$x = 0.6\overline{5}6\overline{5}56\overline{5}556\overline{5}5556\ldots,$$

yet is *non-recurring.*

Having considered the merits and drawbacks of using recurring expansions, and following Pixley's suggestion [62], we have chosen to define the finitely representable numbers to be the set of the reals that have *finite radix-r expansions.* Unfortunately, this choice is radix-dependent — for instance, in radix 9 the number 1/9 is written simply as 0.1, while in radix 10 it requires an infinite sequence of 1's for its representation. It is also not closed under division, as dividing two finite-expansion numbers may yield a number that has an infinite, albeit periodic, expansion. However, the addition, subtraction, and multiplication algorithms for this system are not very difficult to implement as they are essentially those of the usual integer arithmetic.

An alternative, which we did not explore but might have some advantages, is to use rationals as the finitely represented numbers, but support a separate and more direct implementation of them as integer pairs, with conversion to infinite positional form performed when required, e.g. to add numbers of mixed type. Conversion in the opposite direction would be by a suitably adapted version of the "round" function discussed on p. 57.

Chapter 5

Conclusions

In the present thesis we have investigated an arithmetic based upon the representation of computable real numbers by lazy infinite sequences of signed digits in a positional radix-r system. We investigated the properties of numbers thus represented and developed a complete set of algorithms on exact reals in such systems. The foregoing analysis suggests that, notwithstanding the claim made by Boehm and Cartwright, the representation of exact real numbers by lazy infinite sequences in a positional system can lead to reasonably efficient implementations of constructive real arithmetic. In particular, our normalization techniques in multiplication, division, and elementary function evaluation largely overcome what they called the granularity effect (for exact numbers, refer to e.g. Figure 6 which shows the number of integer operations in the multiplication algorithm reduced by a factor of 10). For the first time a method was presented which allowed the summation of series and evaluation of elementary functions.

We have also extended our results to complex numbers represented by sequences of signed real digits written to an imaginary radix ri, and shown that the algorithms remain largely the same, except for normalization. This allows for a more efficient implementation of exact complex numbers and leads one to believe that most of the radix-dependence of an algorithm in a positional system is encoded in the normalization function.

The algorithms have been implemented in the functional programming language Miranda and presented as abstract classes of exact real and complex numbers equipped with a comprehensive package of functions. For the sake of simplicity, our Miranda implementation used $r = 10$ and $\rho = 6$, but there are a number of ways the code can be made more efficient (for instance, choosing the radix to be a large number, and avoiding the unnecessary storage consumption and gratuitous copying of lists). Using a compiled language such as C/C++ or Java would also yield a large dividend in efficiency, which could be improved even further on a multi-processor system if normalization and other functions were multi-threaded. A partial implementation in the C language is also available (see Appendix C), and we have explained how the existing implementation might have its code efficiency improved.

5.1 Pros and cons of positional weighted systems

What are the advantages and shortcomings of positional weighted systems, and what tasks are they best suited for?

Advantages:

- Positional weighted systems seem to be the most natural choice because of the widespread use of the decimal system. Conversion of numbers into redundant form and decoding them back into conventional form is also very simple, as the "conventional" systems are also positional radix-r systems. Conversion is trivial if the radix is a power of 10. Other radices can be used to improve performance and take advantage of the hardware, but conversion to and from decimal is relatively inexpensive.

- Laziness. Each digit gives a better approximation to the number being computed, thus avoiding recomputing the elements calculated earlier. A demand-driven system that uses a positional representation needs only compute those numbers that are needed, and only to the precision required.

- Universality. The same positional algorithms can be expected to work with quite different entities, be it real, complex or (say) p-adic numbers. Almost the only function that needs modifying is the underlying normalization function. This already gives us an advantage when computing with complex numbers in radix ri, but it might also become useful if the code were to be modified to work with other (positionally represented) objects.

Disadvantages:

- Evaluation of transcendental functions in radix-r systems is problematic and very expensive, as there are no obvious digit-by-digit algorithms that are both simple and efficient.

- Expensive space requirements. Any extended sequence of computations that involves exact numbers requires an enormous amount of memory for its completion. Although clever optimization techniques might reduce the extent of the problem (see Section 4.2.4), the memory requirements can be expected to remain on the expensive side.

- The problem of choosing the subset of finitely representable numbers. The availability of a subset in which numbers are represented finitely is important for many reasons, not least of which is the need to compute equality tests. For example, it is clear that all integers must be finitely represented. A possible solution for future exploration would be to use rationals but with their own separate representation, as sketched at the end of Section 4.2.5.

All these factors must be considered prior to choosing a representation of the exact reals most suitable for a given problem. For example, the continued fraction system has some elegant algorithms for transcendental functions, but operations such as addition and subtraction are very expensive.

5.2 Alternatives to exact real arithmetic

- Multiple-precision arithmetic. Multiple-precision computation uses numeric precision beyond the single or double precision ordinarily provided in hardware. Multiprecision arithmetic is significantly more efficient than infinite precision arithmetic, and can be used in cases where a precision level double or triple that of hardware precision is all that is required. One of the best multiple precision arithmetic packages is Bailey's package MPFUN [6], which consists of about 10,000 lines of FORTRAN 77 code in 87 subprograms (complex multiprecision numbers are also supported). Another recent piece of multiple precision software is the FM package of Smith [76], which is functionally similar to Bailey's MPFUN.

- Interval arithmetic. Interval analysis has been an area of extensive research since 1960s. As the name suggests, the idea is to perform arithmetic on intervals $[a, b]$ $(b \geq a)$, and provide as the solution to a problem an interval in which the desired result is guaranteed to lie, thus maintaining bounds on the errors in limited precision calculations for all quantities being computed. We mention one reference, a book by Moore [52], which provides a good introduction to the subject and contains further references. A more recent reference is Alefeld and Herzberger [3].

- Rational arithmetic represents each rational number by two unbounded integers that form its numerator and denominator, thus bypassing the implied division with its sometimes infinite result. The operations on pairs are performed in the obvious way, and one might require them to be relatively prime and the denominator to be positive.

Unfortunately, none of these methods are entirely satisfactory. Multiple precision arithmetic can fail in the same way as standard floating-point, as shown in Section 1.1. Interval arithmetic does not produce exact answers and tends to be overly pessimistic in measuring errors (the error bounds may increase more rapidly than the error itself), while rational arithmetic is not closed under many operations of interest (e.g. square root) and also becomes very slow and inefficient as an extended sequence of computations usually results in very large numerators and denominators.

5.3 Practical applications

There are a number of situations, ranging from the highly theoretical to the completely practical, where one might want to think about infinite precision arithmetic.

One important area of applications of exact real and complex arithmetic is in numerical analysis. For example, it is often desirable to test an algorithm designed for floating-point arithmetic to determine whether it suffers from a numerical instability. This has been traditionally achieved with multiple precision arithmetic, but even multiprecision sometimes fails to produce accurate results. A number of applications of multiple precision arithmetic in numerical analysis and approximation theory are described in [86], and many of these examples also apply to exact arithmetic.

Another possible application of infinite precision arithmetic is in pure mathematics. The fact that a number representation has a precise mathematical meaning has important consequences for the design of programs. We can design functions to precisely satisfy fundamental mathematical relationships (e.g. $\cos^2 x + \sin^2 x = 1$) and prove that our defined functions meet their specifications using conventional methods. Exact computations can be used to explore conjectures and reject those that are invalid. This might be useful in the study of the behaviour of important mathematical constants, such as the classical constants e or π, or the more recent Feigenbaum δ constant [30, 15] and the Bernstein β constant [86]. Infinite precision arithmetic methods can be used to compute some numbers to very high precision, either to see whether there are any statistical anomalies in the results, or to serve as reference values for conventional methods.

5.4 Availability of infinite arithmetic packages

A Miranda version of the package described in this thesis can be found in Appendix A. Later versions will be made available on the University of Kent's Web site, and can be requested directly from the author. A complete C/C++ library for exact real and complex arithmetic is also in the works.

The author knows of several other packages:

- Boehm and Cartwright's constructive reals calculator available from the URL: `http://reality.sgi.com/boehm/calc/`. A Java-based reimplementation was recently completed at SGI and is available from the URL `http://oss.sgi.com/projects/crcalc/`.
- Potts' LFT implementation (Haskell/Java) [26]. Current status unknown.
- David Lester's (Manchester University) Haskell implementation based on Vuillemin's continued fraction algorithms. Current status unknown.
- David Plume's real number calculator based on the *signed binary system*. Available from the URL `http://www.dcs.ed.ac.uk/~dbp/`.

5.5 Conclusion

After the relatively long period of inactivity that followed the publication of Boehm and Cartwright's paper [11], the last three or four years have seen an explosion of research on exact real numbers [24, 51, 28, 25, 26, 63, 75]. Comparing the different approaches in terms of either their theoretical complexity or their performance, would be an interesting area for research, although in the absence of source code, comparing the actual implementations is often next to impossible.

Each of the approaches to exact computing has a number of unresolved problems, yet they all seem to share the unfortunate conclusion that constructive real arithmetic is extremely slow in comparison with hardware floating point. Some measurements have revealed that floating point performs 50 to 100 times faster than infinite precision arithmetic [73].

It seems that the performance problems encountered in exact computations are imposed more by the very nature of real numbers and computers than by the implementations themselves. The main difference between floating point numbers and exact reals is that the former are *self-contained* — once a floating-point number has been computed, all its dependencies can be discarded. Exact numbers, on the other hand, are infinite objects, and must retain all of their dependencies because more precision might be required in the future. Hence, an extended sequence of computations can be expected to generate large dependency graphs, leading to a significant performance slowdown. Another problem with infinite precision arithmetic is that the form of an expression can often affect its computational complexity in unexpected ways (for example, the addition of n numbers is significantly more efficient than $n-1$ nested pairwise additions).

It is therefore doubtful whether exact real arithmetic will ever perform on current architectures as speedily as floating point, which was designed to match the underlying hardware, although one can think of various ways to optimize the existing implementations. There is a lot of room for improvement as far as the algorithms are concerned. Exact arithmetics will perform better on concurrent processing architectures, since many arithmetic operations can be defined in terms of functions which can be executed in parallel (e.g. `reduce`). It is possible that the performance of a highly optimized multi-threaded imperative implementation would be comparable with that of floating point for certain types of computational problems (and in particular, those that do not tend to generate complex sets of dependencies).

Exact arithmetic has important advantages, of unlimited precision and correct mathematical properties, which would make it extremely attractive were it not so expensive. Computers are still doubling in speed and memory capacity approximately every 18 months, and this is predicted to continue for several more orders of magnitude before we hit fundamental physical limits. It is likely that in 2020 computers will be a thousand times faster than today, and will therefore run exact real arithmetic at the speed we get from hardware floating point today, which will make it usable for many applications, for which the then even greater speed of hardware floating point is not required.

This is without assuming any special hardware support for infinite reals. It would in fact be very easy to have direct hardware support for unbounded precision integers and for garbage collection, and both of these things can be expected to happen in the future, as the use of functional and other higher level languages (such as Java) becomes standard. This will reduce, although not eliminate, the performance penalty of exact real arithmetic compared with floating point.

Appendix A

Miranda source code for exact real arithmetic


```
FILE: common.m
--------------
|| Common definitions for the infinite precision package

||----------------------------------------------------------------------||
|| numbers are represented as triples, (type,exponent,mantissa) type    ||
|| is finite or infinite exponent is any integer and mantissa is list   ||
|| of digits in [-6..6] eg (finite,-1,[4,3,5]) represents 0.435         ||
||----------------------------------------------------------------------||

||real == (bool,num,[num])

finite = True
infinite = False

typ (t,e,m) = t
```

```
expo (t,e,m) = e
man (t,e,m) = m

weaker_type x y = typ x & typ y  || infinite if either x or y is infinite

|| machine arithmetic functions (div and rem)
|| the sign of div is positive if a and b have same sign
|| the sign of rem is same as sign of a

divrem a b
  = (d,r),   if a>=0 & b>0
  = (-d,-r), if a<0 & b>0
  = (-d,r),  if a>=0 & b<0
  = (d,-r),  if a<0 & b<0
    where
    (d,r) = (a' div b', a' mod b')
    a' = abs a
    b' = abs b

|| string arithmetic functions

|| absolute value
ab [] = []
ab (a:x) = 0:ab x,       if a=0
         = a:x,          if a>0
         = negate (a:x), otherwise

|| string addition
add (a:x) (b:y) = a+b:add x y
add x [] = x
```

```
add [] y = y

|| string negation
negate = map neg

|| string of zeros
zzz = repeat 0
zeros n = rep n 0

||-------------------------------------------------------------------||
|| If x is either non-zero or a finite rep, then remove_leading_zeros x ||
|| is a rep of the same number and whose mantissa is either [] or has   ||
|| non-zero first element.  In case that the mantissa of x is an        ||
|| infinite list of zeros, remove_leading_zeros x is bottom.            ||
||-------------------------------------------------------------------||
remove_leading_zeros (t,e,m)
  = (t,e,m),                              if m=[]
  = remove_leading_zeros (t,e-1,tl m), if m!0 = 0
  = (t,e,m),                              otherwise

|| kill_zeros n x kills  at most  n leading zeros
kill_zeros n (t,e,m)
  = kill_zeros (n-1) (t,e-1,tl m), if m~=[] & m!0=0
  = (t,e,m),                        otherwise

kill_all_zeros = kill_zeros (-1)

add_leading_zero::[num]->[num]
add_leading_zero list = 0:list
```

```
|| Some useful functions


|| iter computes iterations of a given function, e.g. iter 3 f x = f(f(f(x)))
iter :: num -> (* -> *) -> * -> *
iter m f x = iter (m-1) f (f x),      if m > 0
           = x,                         otherwise


|| The factorial function
fac 0 = 1
fac n = n * faclist!(n-1)

faclist = map fac [0..]
```

```
FILE:  radix10.m
----------------
%include "common.m"

||----------------------------------------------------------------------||
||      Functions specific to radix 10                                  ||
||----------------------------------------------------------------------||

|| Radix-10 normalization functions
reduce [] = []
reduce (a:x)
  = add (a:map fst divrem) (map snd divrem)
    where
    divrem = map f x
    || here r = 10, rho = 6
    f n = (m,d),         if n>=0 & m<6
        = (m-10,d+1),    if n>=0 & m>=6
        = neg2 (f (-n)), otherwise
          where
          d = n div 10
          m = n mod 10
          neg2 (x,y) = (-x,-y)

|| suppose n = (t,e,m) is a numrep with each element of m between -15,15
|| then can n is a numrep for the same number, but with each element of its
|| mantissa between -6,6
can (t,e,m)
  = (t,e,m),                 if m=[]
  = can (t,e+1,reduce (0:m)), if m!0 > 5 \/ m!0 < -5
  = (t,e,reduce m),          otherwise
```

```
|| freduce is the "finite reduce" function; freduce m x normalizes the first
|| m entries of x
freduce m x = reduce (take m x) ++ drop m x

|| times10 multiplies a mantissa by 10
times10 (a:b:x) = 10*a+b:x
times10 [] = []
times10 [a] = [10*a]
```

```
FILE: input.m
-------------
|| Conversion functions in base 10 ( num -> real )

%include "common.m"
%include "radix10.m"

|| Input conversion functions

||--------------------------------------------------------------------||
|| validnum checks whether a string represents a valid number, e.g.    ||
|| that it doesn't contain non-digits                                  ||
||--------------------------------------------------------------------||
validnum :: [char] -> bool
validnum st = ((validate "0123456789+-.Ee" st) & ((int st) \/ (inteint st)
              \/ (intdotposint st) \/ (intdotposinteint st)))

||--------------------------------------------------------------------||
|| validate a b checks whether string b contains any characters other  ||
|| than those that appear in string a.                                 ||
|| Returns True if no other characters appear, False otherwise        ||
||--------------------------------------------------------------------||
validate :: [char] -> [char] -> bool
validate vs st = ((foldr fltr st vs) = [])
                 where
                 fltr a s = filter (~=a) s

||--------------------------------------------------------------------||
|| int checks whether a string represents an integer in the form      ||
|| +/-[digits]                                                         ||
```

```
||------------------------------------------------------------------------||
int :: [char] -> bool
int [] = False
int (a:x)
  = validate "0123456789" x,    if (a='+') \/ (a='-') \/ (digit a)
  = False,                      otherwise

||------------------------------------------------------------------------||
|| inteint checks whether a string represents an integer in the form     ||
|| +/-[digits]e+/-[digits]
||------------------------------------------------------------------------||
inteint :: [char] -> bool
inteint [] = False
inteint st = (member st 'e') & (int (before_e st)) & (int (after_e st))

||------------------------------------------------------------------------||
|| posint checks whether a string represents a positive integer in the   ||
|| form [digits] (plus sign not allowed)                                  ||
||------------------------------------------------------------------------||
posint :: [char] -> bool
posint [] = False
posint st = validate "0123456789" st

||------------------------------------------------------------------------||
|| intdotposint checks whether a string represents an integer in the     ||
|| form  +/-[digits].[digits]                                             ||
||------------------------------------------------------------------------||
intdotposint :: [char] -> bool
intdotposint [] = False
intdotposint st = (member st '.') & (int (before_dot st)) &
```

```
                  (posint (after_dot st))

||----------------------------------------------------------------------||
|| intdotposinteint checks whether a string represents an integer in     ||
|| the form  +/-[digits].[digits]e+/-[digits]                            ||
||----------------------------------------------------------------------||
intdotposinteint :: [char] -> bool
intdotposinteint [] = False
intdotposinteint st = (member st '.') & (member st 'e') &
                      (int (before_dot st)) &
                      (posint (after_dot_before_e st)) & (int (after_e st))

||----------------------------------------------------------------------||
|| A bunch of useful functions used in parsing integers                  ||
||----------------------------------------------------------------------||
before_dot st = takewhile (~='.') st
after_dot st = tl (dropwhile (~='.') st)
before_e st = takewhile (~='e') st
after_e st = tl (dropwhile (~='e') st)
after_dot_before_e = (before_e . after_dot)

||----------------------------------------------------------------------||
|| getexp returns the exponent of a number known to be valid (as         ||
|| previously checked by validnum), in which case it's just the part     ||
|| that follows 'e'.  Here "num" is *unbounded* integer type.            ||
||----------------------------------------------------------------------||
getexp :: [char] -> num
getexp st
   = 0,                                         if ~(member st 'e')
   = convert_char2num (after_e st),             otherwise
```

```
convert_char2num st
   = foldl multadd 0 (map char2digit st),       if digit (hd st)
   = convert_char2num (tl st),                  if (hd st) = '+'
   = neg (convert_char2num (tl st)),            otherwise
     where
     multadd a b = 10*a + b

char2digit a = code a - code '0'

||------------------------------------------------------------------------||
|| getman returns the mantissa of a number known to be valid (as          ||
|| previously checked by validnum).  It is simply found as the part       ||
|| that precedes 'e' (if 'e' is present), and returned as a list of       ||
|| num's, of which the first "num" can be potentially unbounded.          ||
||------------------------------------------------------------------------||
getman :: [char] -> [num]
getman st
   = convert_char2man st,               if (int st) \/ (intdotposint st)
   = convert_char2man (before_e st),    otherwise

|| N.B.: convert_char2man negates the fractional digits if the integer part
|| is negative
convert_char2man st
   = [convert_char2num st],                               if ~(member st '.')
   = (int_part):(map char2digit (after_dot st)),          if (int_part >= 0)
   = (int_part):(map (neg.char2digit) (after_dot st)),    otherwise
     where
     int_part = convert_char2num (before_dot st)
```

```
|| repp x returns a list consisting of repetitions of x.
|| E.g. repp [2,4,3] = [2,4,3,2,4,3,2,4,3,...]
repp :: [num] -> [num]
repp a = a ++ repp a
```

```
FILE:  output.m
---------------
%include "common.m"
%include "radix10.m"

||  Various functions to output exact real numbers

||-------------------------------------------------------------------||
||      Comparison tests among finite objects                        ||
||-------------------------------------------------------------------||
sign (t,e,m)
  = error "attempt to take sign of infinite number", if t=infinite
  = f m,                                              otherwise
    where
    f [] = 0
    f (a:x) =  1,         if a>0
            = -1,         if a<0
            = f x,        if a=0

postest x = (sign x = 1)
negtest x = (sign x = -1)
zerotest x = (sign x = 0)

||-------------------------------------------------------------------||
||      conversion to (sort of) standard representation              ||
||-------------------------------------------------------------------||
|| if x is a -6 to 6 rep, then usual_rep x has each element of the
|| mantissa between -1,9 if x>=0 and between -9,1 if x<=0
usual_rep (t,e,m)
  = (t,e,h m)
```

```
    where
    h [] = []
    h (a:x)
      = a:h x,   if a=0
      = p (a:x), if a>0
      = n (a:x), otherwise
        where
        p [a] = [a]
        p (a:b:x)
          = a:p (b:x),      if b>=0
          = a-1:p (b+10:x), otherwise
        n [a] = [a]
        n (a:b:x)
          = a:n (b:x),      if b<=0
          = a+1:n (b-10:x), otherwise

|| see x is a visual representation of x, where x is a finite rep
see (t,e,m)
  = g (t,e,m),                     if t=finite
  = error "see of infinite rep", otherwise
    where
    g (t,e,m)
      = "0.0e"++numberformat e, if zerotest (t,e,m)
      = '-':h (e,q (negate m)), if negtest (t,e,m)
      = h (e,q m),              otherwise
    q = reverse.carries.cancelzeros.reverse.p
    p [a] = [a]
    p (a:b:x)
      = a: p (b:x),     if b>=0
      = a-1:p (b+10:x), otherwise
```

```
    cancelzeros [] = []
    cancelzeros (a:x)
      = cancelzeros x, if a=0
      = a:x,           otherwise
    carries [] = []
    carries [a] = [a]
    carries (a:b:x)
      = a:carries (b:x),                        if a>=0
      = 9:carries (b-1:x),                      if a = -1
      = error ("carries ("++shownum a++":...)"), otherwise
    h (e,m)
      = h (e-1,tl m),                 if m!0=0
      = f m ++ "e" ++ numberformat e, otherwise
        where
        f [a] = mkdigit a:".0"
        f (a:b:x) = mkdigit a:'.':map mkdigit (b:x)

mkdigit n = "0123456789"!n

numberformat x = numrep x []

numrep n y
  = "(-" ++ f (-n) y ++ ")", if n<0
  = f n y,                   otherwise
    where
    f n y
      = mkdigit n:y,                          if n<10
      = f (n div 10) (mkdigit (n mod 10):y), otherwise
```

```
|| truncate n x is a finite number rep which differs from x by less than
|| 10^n - so to see more, make n smaller (eg -10)
truncate n (t,e,m)
  = (finite,e',m')
    where
    (e',m')
      = (e, take(e-n+1) m), if n<=e
      = (e,[]),             otherwise

seetruncated n = see.truncate n

funman :: [num] -> [char]
funman (a:x)
   = (shownum a) ++ funman x,   if a ~= -1
   = "I" ++ funman x,           otherwise
funman [] = []
```

```
FILE:  reals.m
--------------
%export real plus minus mult divby negof div_by_2 sqroot eexp elog esin
        ecos etan ecot atan absn min_max power pie ee zero one two three
c nat st cf cr fun

abstype real                                    || abstract real type
with plus, minus, mult, divby :: real -> real -> real
     negof, div_by_2, sqroot, eexp, elog,
     esin, ecos, etan, ecot, atan, absn :: real -> real
     min_max :: real -> real -> (real,real)
     serieslimit :: [real] -> real
     power :: real -> num -> real               || power x n = x^n
     pie, ee, zero, one, two, three :: real     || some useful constants
     c :: num -> real                           || convert a float to real
     nat :: num -> real                || convert a natural number to real
     cf :: [char] -> real              || convert unbounded float to real
     cr :: [char] -> [char] -> real    || convert repeating float to real
     st :: num -> real -> [char]       || show truncated number
     showreal :: real -> [char]        || default precision = 10 digits
     fun :: real -> [char]             || online output function

||---------------------------------------------------------------------||
|| numbers are represented as triples, (type,exponent,mantissa) type    ||
|| is finite or infinite exponent is any integer and mantissa is list   ||
|| of digits in [-6..6] eg (finite,-1,[4,3,5]) represents 0.435         ||
||---------------------------------------------------------------------||
real == (bool,num,[num])

%include "common.m"     || common definitions
```

```
%include "radix10.m"    || radix-10 specific functions
%include "input.m"      || input conversion functions
%include "output.m"     || output functions

||*********************************************************************||
|| Addition, subtraction, multiplication, division, and absolute value  ||
||*********************************************************************||

||---------------------------------------------------------------------||
||                      plus (addition function)                        ||
|| If x, y are -6 to 6 numreps then (plus x y) is a -6 to 6 numrep of   ||
|| their sum.   (plus x y) is infinite iff either x or y is infinite.   ||
||---------------------------------------------------------------------||
plus x y
  = can (t,e,m), if expo x >= expo y
  = plus y x,    otherwise
    where
    t = weaker_type x y
    e = expo x
    m = add (man x) (man y),                          if expo x = expo y
      = add (man x) (zeros (expo x-expo y) ++ man y), otherwise

||---------------------------------------------------------------------||
||                  minus (subtraction function)                        ||
|| If x, y are -6 to 6 numreps then (minus x y) is a -6 to 6 numrep of  ||
|| (x-y).   (minus x y) is infinite iff either x or y is infinite.      ||
||---------------------------------------------------------------------||
minus x (t,e,m) = plus x (t,e,negate m)
```

```
|| negof x is negative of x
negof (t,e,m) = (t,e,negate m)


||----------------------------------------------------------------------||
||                    mult (multiplication function)                    ||
|| If x, y are -6 to 6 numreps then (mult x y) is a -6 to 6 numrep of   ||
|| their product x*y.   (mult x y) is inf iff either x or y is inf.     ||
||----------------------------------------------------------------------||
mult x y
  = can (t,e,m)
    where
    t = weaker_type x y
    e = expo x + expo y
    m = mantissa_mult (man x) (man y)
    mantissa_mult xm ym
      = [],                               if xm=[] \/ ym=[]
      = mult_reduce cross_products_list,  otherwise
        where
        cross_products_list = [map (* t) ym | t<-xm]
        mult_reduce z = [],                                     if z=[[]]
            = take 26 block ++
                      mult_reduce (drop 26 block : drop 27 z),  otherwise
              where
              block = (reduce.reduce.diag_add.take 27) z
              diag_add (a:b:x)
                = [],                               if a=[]
                = a!0 : diag_add (add (tl a) b : x), otherwise
              diag_add [a] = a
              diag_add [] = []
```

```
|| The following function performs multiplication by 10 (unnormalized)
mult10 (a:b:x) = 10*a+b:x
mult10 [] = []
mult10 [0] = []
mult10 [a] = [10*a]

||----------------------------------------------------------------------||
||                      divby (division function)                       ||
|| If x, y are -6 to 6 numreps, val x ~= 0, then (divby x y) is a       ||
|| -6 to 6 numrep of (y/x).                                             ||
||----------------------------------------------------------------------||
divby x y ||x,y are -6 to 6 numreps, VAL x ~= 0
  = (can.kill_zeros 3) (infinite, exponent, repdiv 1 (man y))
    where
    exponent = expo y - expo divisor
      || divisor is a rep having same val as x such that
      || 100 <= abs divisor <= 996 & tl (man divisor) is a -6 to 6 rep
      || except that if VAL x = 0, then divisor is undefined
    divisor = norm_divisor (kill_all_zeros x)
    norm_divisor (t,e,m)
      = error "attempt to divide by zero", if m=[]
      || m!0 ~= 0, by construction
      = norm_divisor (t,e-1,[10*m!0]),             if abs (m!0) < 100 &
                                                      tl m = []
      = norm_divisor (t,e-1,10*m!0+m!1:tl(tl m)),  if abs (m!0) < 100 &
                                                      tl m ~= []
      = (t,e,m),                                   otherwise
    (d:ds) = man divisor
    divisor_multiples = [map (*(-k)) ds | k<-[-10..10] ]
    repdiv counter n
```

```
      = [],                 if n=[]
      = nextdigit:quotient, otherwise
        where
        (nextdigit,leftover) = divrem (n!0) d
      || following Pixley's suggestion, we normalize only 8 leading digits,
      || except every 10 terms, we normalize a little more (20 digits), and
      || every 100 terms, we normalize fully
        quotient
          = repdiv 1 (reduce (mult10 m)),   if counter mod 100 = 0
          = repdiv (counter+1) remainder1,  if counter mod 10 = 0
          = repdiv (counter+1) remainder2,  otherwise
            where
            remainder1 = reduce (mult10 (take 20 m)) ++ drop 20 m
            remainder2 = reduce (mult10 (take 8 m)) ++ drop 8 m
        m = leftover:add (tl n) (divisor_multiples!(nextdigit+10))

||-------------------------------------------------------------------------||
||                  div_by_2 (division by 2 function)                      ||
||  If x is a -6 to 6 numrep, then div_by_2 x is a -6 to 6 numrep of      ||
||  x/2, having the same type and exponent.                                ||
||-------------------------------------------------------------------------||
div_by_2 x
  = can (typ x, expo x, d2 (man x))
    where
    d2 [] = []
    d2 x
      = add (map fst k) (0:map snd k)
        where
        k = map f x
        f t = (t div 2, 0),     if t mod 2 = 0
```

```
            = (t div 2, 5),     otherwise

||-----------------------------------------------------------------------||
||                        absn (absolute value)                          ||
||             absn is a -6 to 6 numrep preserving function              ||
||-----------------------------------------------------------------------||
absn x = (typ x, expo x, ab (man x))

||-----------------------------------------------------------------------||
||   min_max takes two -6 to 6 reps and returns a pair of -6 to 6 reps, ||
||   being the min and max of x,y repectively.  Both results are of     ||
||   type infinite if either input is infinite.                          ||
||-----------------------------------------------------------------------||
min_max x y
  = (div_by_2 (minus a b) , div_by_2 (plus a b))
    where
    a = plus x y
    b = absn (minus x y)

||-----------------------------------------------------------------------||
||                   sqroot (square root function)                       ||
||-----------------------------------------------------------------------||
sqroot (t,e,x)
  = (t,e,x),                      if (x = []) \/ (x = [0])
  = can (infinite,exponent,y),    otherwise
    where
    exponent = e div 2
    y = manroot x,              if e mod 2 = 0
      = manroot (times10 x),    otherwise
```

```
||manroot::[num]->[num]
manroot [] = repeat 0
manroot x
  = add_leading_zero (manroot (tl (tl x))),  if (tl x ~= []) & (x!0 = 0) &
                                                (x!1 = 0)
  = add_leading_zero ylist,                  otherwise
    where
    ylist = repeat 0,                     if (xlist = []) \/ (xlist!0 = 0)
          = error "attempt to take square root of a negative number",
                                  if xlist!0 < 0
          = entier(sqrt (xlist!0)) : (reproot 1 first),
                                  otherwise
    xlist = times10 (times10 x)
    first = times10 ((xlist!0-(ylist!0)*(ylist!0)) : (tl xlist))
    reproot counter remainder
      = digit : reproot (counter+1) nextremainder
        where
        (digit,leftover) = divrem (remainder!0) (2*ylist!0)
        nextremainder
  = freduce counter (times10 (leftover :
                           add (tl remainder) (negate modifier)))
        modifier = (map (*2*digit) (take (counter-1) (tl ylist))) ++
                            [digit*digit]

|| power x n computes the n-th power of x
power x n
  = (finite,0,[1]),             if n = 0
  = x,                          if n = 1
  = mult pow_x_nd2 pow_x_nd2,   if n mod 2 = 0
  = mult x (powersx!(n-1)),     otherwise
```

```
    where
    pow_x_nd2 = powersx!(n div 2)
    powersx = powers x

powers x = map (power x) [0..]

|| series limits
serieslimit (a:x)
  = can (infinite, largest_exp (expo a) x, serieslimitman (a:x))
    where
    largest_exp e (a:x)
       = e,                        if e > (expo a)
       = largest_exp (expo a) x,   otherwise

serieslimitman (a:b:x)
   = serieslimitman ((ppp (takewhile (equalexp ea) (a:b:x))):
                    (dropwhile (equalexp ea) (a:b:x))),   if ea = eb
   = (take' (ea-eb) mana) ++ serieslimitman ((typ a, eb,
                    drop (ea-eb) mana):b:x),              if ea > eb
   = serieslimitman ((pluss a b):x),                      otherwise
     where
     equalexp a x = (a = (expo x))
     mana = man a
     ea = expo a
     eb = expo b

take' n (a:x) = a:take' (n-1) x, if n>0
              = [],              otherwise
take' n [] = zeros n, if n>0
           = [],      otherwise
```

```
pluss (t,e,m) (tt,ee,mm)
  = (t & tt, e,reduce (add m (zeros (e-ee) ++mm))),      if e >= ee
  = (t & tt,ee,reduce (add(zeros (ee-e)++m)mm)),         otherwise

|| ppp only adds arguments with the same exponent!
ppp :: [(bool,num,[num])] -> (bool,num,[num])
ppp x = (typ (hd x),expo (hd x),norm (#x) (vector_add (map man x)))

vector_add :: [[num]] -> [num]
vector_add [] = []
vector_add (a:b:x)
   = vector_add ((add a b):x),          if  x ~= []
   = add a b,                           otherwise

norm :: num -> [num] -> [num]
norm n x
  = iter m reduce x
    where
    m = (entier ((log (6*n-5))/(log 10))) + 1


|| Computation of Pi using Ramanujan's pi formula
pie = divby inverse_pi one
inverse_pi = mult (divby (nat 9801) (mult two (sqroot two)))
                  (serieslimit ramanujan)
ramanujan = [ divby (nat (((fac k)^4)*(396^(4*k))))
              (nat (((fac (4*k))*(1103+26390*k)))) | k <- [0..] ]
```

```
||------------------------------------------------------------------------||
||                       Elementary functions                             ||
||------------------------------------------------------------------------||

|| esin uses the following algorithm - first, we compute the number modulo
|| 2*pi, i.e. guess the number of periods (at least approximately - say,
|| 3 most significant digits), then multiply pi by the number of periods,
|| subtract it from the original number, and compute the sine of that.
esin x
  = zero,                                   if man x=[]
  = esin' (remove_leading_zeros x'),        otherwise
    where
    x' = minus x (mult (nat periods) two_pi)
    two_pi = mult2 pie
    periods = int_part (divby two_pi x)

|| esin' assumes a normalized argument
esin' x = serieslimit [ times_neg_one k (divby (nat (fac (2*k+1)))
                        (power x (2*k+1))) | k <- [0..] ]

|| similarly for ecos
ecos x
  = one,                                    if man x=[]
  = ecos' (remove_leading_zeros x'),        otherwise
    where
    x' = minus x (mult (nat periods) two_pi)
    two_pi = mult2 pie
    periods = int_part (divby two_pi x)

ecos' x = serieslimit [ times_neg_one k (divby (nat (fac (2*k)))
```

```
          (power x (2*k))) | k <- [0..] ]

|| functions used by esin and ecos

|| multiplies a number by 2
mult2 (t,e,m) = can (t,e, map (*2) m)

|| returns the integer part of the number
int_part :: (bool,num,[num]) -> num
int_part (t,e,m)
   = 0,                                             if e < 0
   = hd m,                                          if e = 0
   = int_part (t,e-1,[10*m!0]),                     if tl m = []
   = int_part (t,e-1,10*m!0+m!1:tl(tl m)),          if tl m ~= []

|| times_neg_one k x computes x * (-1)^k
times_neg_one :: num -> real -> real
times_neg_one k x
   = x,          if k mod 2 = 0
   = negof x,    otherwise

|| Tangent
etan x = divby (ecos x) (esin x)

|| Cotangent
ecot x = divby (esin x) (ecos x)

|| Arctangent
atan x = serieslimit [ times_neg_one k (divby (nat (2*k+1))
                      (power x (2*k+1))) | k <- [0..] ]    ||  |x| < 1  !!!
```

```
|| Natural logarithm
|| elog x works for all x > 0
elog x = mult two (serieslimit [ divby (nat (2*k-1))
                                       (power y (2*k-1)) | k <- [1..] ])
         where
         y = divby (plus x one) (minus x one)

|| eexp checks whether the exponent is > 0, and if it is,
|| computes exp x as exp (10^n) * exp (man)
eexp (t,e,m)
   = exp_power_series (t,e,m),                         if e <= 0
   = power (exp_power_series (t,0,m)) (10^e),          otherwise
     where
     exp_power_series x = serieslimit [ divby (nat (fac k))
                                              (power x k) | k <- [0..] ]

|| ee ("exact e") is computed using David Turner's factorial base algorithm
ee = can (infinite, 0, edigits)
     where
     edigits = 2 : convert (repeat 1)
     convert x = hd x':convert (tl x')
                 where x' = enorm 2 (0:map (10*) x)
     enorm c (d:e:x) = d + e div c: e' mod c : x', if e mod c + 9 < c
                     = d + e' div c : e' mod c : x', otherwise
                       where
                       (e':x') = enorm (c+1) (e:x)
|| input functions
c x = cf (show x)
```

```
||------------------------------------------------------------------------||
|| cf converts a string of chars representing a floating-point number    ||
|| in the form +/- (integer part) . (fractional part) E/e +/- (int exp)  ||
|| to exact real form in base 10                                          ||
||------------------------------------------------------------------------||
cf st
   = can (True, getexp sst, getman sst),     if (validnum sst)
   = error "Invalid number",             otherwise
     where
     sst = map lowercase (filter (~=' ') st)   || filter out all spaces
     lowercase a =  a ,     if a ~= 'E'        || and change 'E' to 'e'
                 = 'e',     otherwise

||------------------------------------------------------------------------||
|| cr converts a string of chars representing a repeating floating-point ||
|| number to exact real form in base 10.  The number is expected as      ||
|| two strings st and repst, intepreted as follows:                      ||
|| st = +/- (integer part) . (fractional part) E/e +/- (int exp)         ||
|| repst = (repeating part).  The number is then taken as:               ||
|| number = +/- (int part).(frac part)(repeating part)E/e +/-(int exp)   ||
|| E.g. 12.3456565656... = cr "1.234e1" "56" = cr "1.2345e1" "65"        ||
||------------------------------------------------------------------------||
cr st repst
   = can (True, getexp sst, repman),    if (validnum sst) &
                                           (validate "0123456789" repst)
   = error "Invalid number",            otherwise
     where
     repman = (getman sst) ++ repp (map char2digit repst)
```

```
     sst = map lowercase (filter (~=' ') st)   || filter out all spaces
     lowercase a =  a ,     if a ~= 'E'        || and change 'E' to 'e'
                 = 'e',     otherwise

||converts a natural number into an exact real
nat n = can (finite,e,m)
        where
        e=#m-1
        m=(reverse.turnnat) n
        turnnat n = (n mod 10):turnnat (n div 10),      if n ~= 0
                  = [],                                 otherwise

|| output functions
st n = seetruncated (-n)
showreal x = st 10 x

|| fun output function (prints the "almost standard" representation online)
|| prints "I" instead of (-1)
fun x = "e"++(shownum e)++" * "++(shownum (m!0))++"."++(funman (tl m))
        where
        (t,e,m) = usual_rep x

|| Some useful numbers
zero = (finite,0,[0])
one = (finite,0,[1])
two = (finite,0,[2])
three = (finite,0,[3])
```

```
SAMPLE OUTPUT OF THE PACKAGE
----------------------------
                     T h e   M i r a n d a   S y s t e m
                   version 2.018 last revised 6 January 1999
                    Copyright Research Software Ltd, 1990
(5000000 cells)
reals.m

Miranda sqroot two
1.4142135624e0
Miranda st 200 $$
1.41421356237309504880168872420969807856967187537694807317667973799073247
46210703885038753432764157273501384623091229702492483605585073721264412149
7099935831413222665927505592755799950501152782060571 47e0
Miranda mult (sqroot two) (sqroot two)
2.0e0
Miranda st 200 $$
2.0e0

Miranda sqroot three
1.7320508076e0
Miranda div_by_2 $$
8.660254038e(-1)
Miranda mult $$ (esin one)
7.287352494e(-1)
Miranda st 100 $$
7.287352493911478103696692984490167347459106606486613115974475831721467930
51948947233282686279888 1088e(-1)

Miranda divby (sqroot (nat 1000))  (cf "3783327123768.832178923189")
```

```
1.19639308446032670975 3e11
```

```
Miranda st 200 $$
1.19639308446032670975314146320059900417310563312350057482905152585079567 8
20306419869444067336627144338064099095473368488556455375059134664540022419
913025847751639495643182977913667279101430146491281104763271514 57e11
```

```
Miranda sqroot (c 9876543)
3.1426967718824e3
Miranda st 1000 $$
3.1426967718823908194707255836795750111145341183174155647816462937222 26745
720935813431674509696248308480039074210068134492236956728282307875766 72163
331187134104556487706244955758241455805015237514911214076194952074629 50171
411423688508849124501557301398340283417729002881504150548720641032097 24946
015067705699579929513044134486581381968117533405333697392123941475892 76170
994495127586956591906630586715372375097078305648566581663368904832847 62168
055726571583474786620755174245559670489175815953249046686323320967132 67775
012229744983498413167562326475460126281743541237278361620524776041907 24616
626357774147676551202884407553584578062892498682645783299294929602110 99488
274467498298348460640838254909039564788900495381087573040362876857319 88407
117901951769577873554010789961279879653155718632583217061889765265956 91174
611713472499107746722640527899717224272801507849665806657395863948059 07059
308163476568511083478396686133201426693462535927864296208129085395839 52113
1880781353510445640580680894508110685091353e3
```

```
Miranda ee
2.7182818284e0
Miranda st 300 $$
2.718281828459045235360287471352662497757247093699959574966967627724 076630
353547594571382178525166427427466391932003059921817413596629043572900 33429
526059563073813232862794349076323382988075319525101901157383418793070 21540
```

```
8914993488416750924476146066808226480016847741185374234544243710753907774
992069e0

Miranda eexp two
7.3890560989e0
Miranda fun $$
e1 * 0.73890561I8930650227230427460575007813180315570551847324087127822522
5738I607905776338431248507912179477375316126547^C<<...interrupt>>

Miranda pie
3.1415926536e0
Miranda st 100 $$
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706
8e0

Miranda esin two
9.092974268e(-1)
Miranda ecos two
-4.161468365e(-1)
Miranda plus (power (esin two) 2) (power (ecos two) 2)
1.0e0
Miranda fun $$
e0 * 1.0000000000000000000000000000000000000000000000^C<<...interrupt>>

Miranda eexp (c 0.5)
1.6487212707e0
Miranda elog $$
5.0e(-1)
Miranda st 100 $$
5.0e(-1)
```

Appendix B

Miranda source code for exact complex arithmetic

```
/*  Copyright (C) Alexander Kaganovsky 1995-1999.  All rights reserved. */

FILE:  complex.m
----------------

%export complex mc cf cr re im cadd csub cmul cdiv showc fun st

%include "input.m"

|| This is different from the previous package in that 100 is used as
|| the exponent base of a complex number, instead of 10.
|| This complicates conversion, but simplifies addition which we think
|| is computationally more important

abstype complex
with mc :: real -> real -> complex
     cf :: [char] -> real
     cr :: [char] -> [char] -> real
```

```
     re, im :: complex -> real
     cadd, csub, cmul, cdiv :: complex -> complex -> complex
     showc :: num -> num -> complex -> [char]
     showcomplex :: complex -> [char]
     fun :: complex -> (num,[num])

complex == (num,[num])
real == (num,[num])

expo (e,m) = e
man (e,m) = m

||------------------------------------------------------------------||
|| Conversion functions (base 10 to 100)                            ||
||------------------------------------------------------------------||
cm10to100 :: [num] -> [num]
cm10to100 [] = []
cm10to100 (a:x)
  = (10*a+(hd x)) : cm10to100 (tl x),   if x ~= []
  = 10*a : cm10to100 x,                 otherwise

|| cman10to100 is the same as cm10to100, except it doesn't convert
|| the first element of the list
cman10to100 (a:x) = a:cm10to100 x
cman10to100 [] = []

||------------------------------------------------------------------||
|| Conversion functions (base 100 to base 10)                       ||
||------------------------------------------------------------------||
cm100to10 :: [num] -> [num]
```

```
cm100to10 [] = []
cm100to10 (a:x)
  = a0 : a1 : cm100to10 x
    where
    (a0,a1) = divrem a 10

cman100to10 (a:x) = a:cm100to10 x
cman100to10 [] = []

||----------------------------------------------------------------------||
|| mc converts two exact reals (base 10) to a single-component          ||
|| complex format (base 10i)                                            ||
||----------------------------------------------------------------------||
mc (ex,mx) (ey,my)
  = (ez, mz)
    where
    mz = alternate (cman10to100 mx') (cman10to100 my')
    (ez,mx',my')
        = (ex div 2, mx, inc (ex-ey-1) my),           if (ex > ey) &
                                                            (ex mod 2 = 0)
        = ((ex-1) div 2, dec mx, inc (ex-ey-2) my),   if (ex > ey+1) &
                                                    (ex mod 2 = 1)
        = ((ex-1) div 2, dec mx, dec my),             if (ex = ey+1) &
                                                    (ex mod 2 = 1)
        = (ey div 2, inc (ey-ex) mx, dec my),         if (ex <= ey) &
                                                    (ey mod 2 = 0)
        = ((ey+1) div 2, inc (ey-ex+1) mx, my),       otherwise

|| dec adjusts the mantissa of a radix-10 number to compensate for a dec of
|| (subtraction of one from) its exponent by multiplying the mantissa by 10
```

```
dec :: [num] -> [num]
dec [] = []
dec [a] = [10*a]
dec (a:x) = (10*a+(hd x)):(tl x)

|| inc adjusts the mantissa of a radix-10 number to compensate for an
|| increase of its exponent by n.  It does so by dividing the mantissa
|| by 10^(-n), i.e. adding n leading zeros
inc :: num -> [num] -> [num]
inc 0 x = x
inc n x = zeros n ++ x
zeros n = rep n 0

||--------------------------------------------------------------------||
|| re returns the real part (represented in base 10) of a complex     ||
|| number (base 10i), im - the imaginary part of same                 ||
||--------------------------------------------------------------------||
re (e,x) = can (2*e, cman100to10 (even x))
im (e,x) = can (2*e-1, cman100to10 (odd x))

||--------------------------------------------------------------------||
|| alternate (x0,x1,...) (y0,y1,...) = (x0,-y0,-x1,y1,x2,-y2,...)     ||
||--------------------------------------------------------------------||
alternate :: [num] -> [num] -> [num]
alternate [] [] = []
alternate [] [y0] = [0,-y0]
alternate [] [y0,y1] = [0,-y0,0,y1]
alternate [x0] [] = [x0]
alternate [x0] [y0] = [x0,-y0]
alternate [x0] [y0,y1] = [x0,-y0,0,y1]
```

```
alternate [x0] (y0:y1:ys) = [x0,-y0,0,y1] ++ alternate [0] ys
alternate [x0,x1] [] = [x0,0,-x1]
alternate [x0,x1] [y0] = [x0,-y0,-x1]
alternate (x0:x1:xs) [y0] = [x0,-y0,-x1,0] ++ alternate xs [0]
alternate (x0:x1:xs) (y0:y1:ys)
  = [x0,-y0,-x1,y1] ++ alternate xs ys

||----------------------------------------------------------------------||
|| even (z0,z1,z2,...) = (z0,-z2,z4,...)                                ||
|| odd  (z0,z1,z2,...) = (-z1,z3,-z5,...)                               ||
||----------------------------------------------------------------------||
even z = skipevenodd 0   1  z
odd  z = skipevenodd 1 (-1) z

skipevenodd :: num -> num -> [num] -> [num]
skipevenodd counter sign (z0:z)
 = (sign*z0):(skipevenodd 1 (-sign) z),        if counter mod 2 = 0
 = skipevenodd 0 sign z,                       otherwise
skipevenodd counter sign [] = []

||----------------------------------------------------------------------||
|| complex normalization functions                                      ||
||----------------------------------------------------------------------||
creduce z = alternate (reduce (even z)) (reduce (odd z))

||ccan :: complex -> complex
ccan (e,m)
  = (e,m),              if m=[]
  = (e,creduce m),       otherwise
```

```
|| Complex addition function
cadd (ez,mz) (ew,mw)
  = ccan (ez,m),                        if ez >= ew
  = cadd (ew,mw) (ez,mz),       otherwise
    where
    m = add mz mw,                                    if ez = ew
      = add mz (zeros (2*(ez-ew)) ++ mw),             if (ez-ew) mod 2 = 0
      = add mz (zeros (2*(ez-ew)) ++ (map neg mw)),   otherwise

add (a:x) (b:y) = a+b : add x y
add x [] = x
add [] y = y

csub (ez,mz) (ew,mw) = cadd (ez,mz) (ew, negate mw)

|| mult z w - product of two complex numbers
cmul z w
  = ccan (e,m)
    where
    e = expo z + expo w
    m = mantissa_mult (man z) (man w)
    mantissa_mult zs ws
      = [],                             if zs=[] \/ ws=[]
      = mult_reduce cross_products_list,  otherwise
        where
        cross_products_list = [map (* t) ws | t<-zs]
        mult_reduce z = [],                                 if z=[[]]
            = take 228 block ++
               mult_reduce (drop 228 block:drop 229 z), otherwise
              where
```

```
              block = (creduce.creduce.diag_add.take 229) z
              diag_add (a:b:x)
                = [],                                if a=[]
                = a!0 : diag_add (add (tl a) b : x),  otherwise
              diag_add [a] = a
              diag_add [] = []


||------------------------------------------------------------------------||
||  complex division                                                      ||
||------------------------------------------------------------------------||
cdiv z w
  = (ccan.kill_zeros 4) (exponent, repdiv (man z))
    where
    exponent = expo z - expo divisor
    divisor = normalize (kill_all_zeros w)
    w0:ws = man divisor
    repdiv z
      = [],                 if z=[]
      = nextdigit:quotient, otherwise
        where
        nextdigit = guess (z!0) ((z!1) div 10) w0 ((ws!0) div 10)
        quotient = repdiv (creduce (times_ri m))
        m = add z (map (*(-nextdigit)) (w0:ws))
        guess z0 z1 w0 w1 = fst (divrem (z0*w0+z1*w1) (w0*w0+w1*w1))

|| kill zeros
kill_zeros n (e,m)
  = kill_zeros (n-2) (e-1,map neg (tl (tl m))), if (tl m)~=[] & m!0=0 &
                                                     m!1=0
```

```
  = (e,m),                                     otherwise

kill_all_zeros = kill_zeros (-1)

add_leading_zero (e,m) = (e,0:m)

|| normalize - normalizes divisor
normalize (e,m)
  = error "attempt to divide by zero",                 if m=[]
  = normalize (e-1,map neg ((times_ri.times_ri) m)),
                     if (abs (m!0) < 10000) & (abs (m!1) < 10000)
  = (e,m),                                             otherwise

|| times_ri multiplies a list by 10i
times_ri [] = []
times_ri [0] = []
times_ri [z0] = [0,-100*z0]
times_ri [z0,z1] = [z1,-100*z0]
times_ri [z0,z1,z2] = [z1,z2-100*z0]
times_ri (z0:z1:z) = z1:((hd z)-100*z0):(tl z)

|| output functions
showc ni nr z
    = (st nr (re z)) ++ " + i * " ++ (st ni (im z))
showcomplex z = showc 10 10 z

fun (x,y) = (x,y)
```

Appendix C

C source code for exact real arithmetic


```
FILE:  reals.h
--------------
/* This file contains all data structures and function definitions */

/*  Each rnode represents a lazy list of digits.   rnodes are   */
/*  tagged by "tlist" or "tapp" representing list nodes and     */
/*  application nodes, respectively.                            */

struct rnode     /* real number node */
{
    enum  { tlist, tapp } tag;
    union
    {
           struct   /* tlist */
           {
                  int           Rhd;    /* head of the list */
```

```
                struct rnode *Rtl;    /* tail of the list */
            } ulist;

            struct   /* tapp */
            {
                void (*Rfun) ();       /* function to apply */
                struct rnode **Rarg;   /* pointer to (list of) arg(s) */
            } uapp;

     } Ru;
};

/*  some useful definitions based upon the rnode struct */
#define hd Ru.ulist.Rhd         /* head of a list */
#define tl Ru.ulist.Rtl         /* tail of a list */

#define fun Ru.uapp.Rfun        /* application of a function */
#define arg Ru.uapp.Rarg        /* its argument(s) */

/*  An exact real number contains the following information:    */
/*  whether it's finite or infinite, the number's integer       */
/*  exponent, pointer to the list of digits (rnode),            */
/*  a reference count, and a list of its dependencies           */
/*  (NULL if it doesn't exist)                                  */
struct real
{
    enum { inf, fin, rep }  type;       /*    type      */
    int *expo;                          /*   exponent   */
    struct rnode *man;                  /*   mantissa   */
```

```
    int ref;                            /*  ref count   */
    struct real **deps; /* dependencies */
};


/* memory allocation */

#define new_rnode() ((struct rnode *)malloc(sizeof (struct rnode)))
#define new_int() ((int *)malloc(sizeof (int)))
#define new_real() ((struct real *)malloc(sizeof (struct real)))
```

```
FILE:  reduce.c
--------------
/*  An implementation of reduce and related functions  */


#include <stdio.h>
#include "reals.h"


void eval (struct rnode *x)
{


/* if x is a list, we leave it alone; otherwise, we evaluate one term */


    if (x->tag == tapp)
       (*x->fun) (x, x->arg);
}


void print (struct rnode *x)
{
    setbuf(stdout, NULL);


    for (; x->tl != NULL ; x = x->tl)
    {
       eval(x);
       printf("%d ", x->hd);
    }
}


/* mult10 x = map (*10) x */
void mult10 (struct rnode *x, struct rnode **farg)
{
```

```
    struct rnode *tmp;

    eval(farg);

    x->tag = tlist;
    x->hd = (farg->hd)*10;

    tmp = x->tl = new_rnode();
    tmp->tag = tapp;
    tmp->fun = mult10;
    tmp->arg = farg->tl;
}

void fadd (struct rnode *x, struct rnode *farg[])
{
    struct rnode *tmp1, *tmp2;
    struct rnode **args;

    args = (struct rnode **) malloc (2 * sizeof (struct rnode *));

    eval(farg[0]); eval(farg[1]);

    tmp1 = new_rnode();
    tmp1->tag = tlist;
    tmp1->hd = (farg[0]->hd) + (farg[1]->hd);

    tmp2 = tmp1->tl = new_rnode();
    tmp2->tag = tapp;
    tmp2->fun = fadd;
```

```
    args[0] = farg[0]->tl; args[1] = farg[1]->tl;
    tmp2->arg = args;

 /* farg was previously allocated by malloc() to hold the arguments of
    a tapp rnode, which we have just replaced with a tlist node.
    Therefore, we can free farg.  */
    free(farg);

    *x = *tmp1;
    free(tmp1);
}

struct rnode * add (struct rnode *a, struct rnode *b)
{
    struct rnode *tmp;
    struct rnode **args;

    args = (struct rnode **) malloc (2 * sizeof (struct rnode *));

    args[0] = a;  args[1] = b;

    tmp = new_rnode();
    tmp->tag = tapp;
    tmp->fun = fadd;
    tmp->arg = args;

    return tmp;
}

/* stuff for reduce */
```

```
int reduce_fst (int x)
{
    int xmod10;

    if (x < 0)
       return (-reduce_fst (-x));

    xmod10 = mod (x,10);

    if (xmod10 < 6)
       return (xmod10);
    else
       return (xmod10 - 10);
}

int reduce_snd (int x)
{
    int xmod10, xdiv10;

    if (x < 0)
       return (-reduce_snd (-x));

    xmod10 = mod (x,10);
    xdiv10 = div (x,10);

    if (xmod10 < 6)
       return (xdiv10);
    else
       return (xdiv10 + 1);
}
```

```
/*
   reduce operates on _infinite_ non-repeating lists, i.e. rnodes are not
   allowed to loop (e.g., a->b->c->a is not allowed).  This is because reduce
   modifies the list it is _applied to_, and does _not_ create a new list.
   Since repeating lists do not expand beyond the last entry, reduce will
   fail trying to write beyond the last rnode.

   If we have an rnode with tag=tapp, fun=reduce, and arg=the number to be
   normalized:

   |--------|         |--------|         |--------|         |--------|
   | tapp   |     |-> | tlist  |     |-> | tlist  |     |-> |   ?    |
   |--------|     |   |--------|     |   |--------|     |   |--------|
   | reduce |     |   |   a    |     |   |   b    |     |   |   ?    |
   |--------|     |   |--------|     |   |--------|     |   |--------|
   | arg ---------    |  tl a -------    |  tl b -------    |   ?    |
   |--------|         |--------|         |--------|         |--------|

   reduce does the following:

   |--------|         |--------|         |--------|         |--------|
   | tlist  |     |-> | tapp   |     |-> | tlist  |     |-> |   ?    |
   |--------|     |   |--------|     |   |--------|     |   |--------|
   |a+snd(b)|     |   | reduce |     |   | fst(b) |     |   |   ?    |
   |--------|     |   |--------|     |   |--------|     |   |--------|
   | tl a --------    |  arg --------    |  tl b -------    |   ?    |
   |--------|         |--------|         |--------|         |--------|
*/
```

```
void reduce (struct rnode *x, struct rnode *farg)
{
    struct rnode *y;
    int tmp1, tmp2;
    struct rnode *args[2];

    args[0] = farg;
    eval (args[0]);

    args[1] = (args[0])->tl;
    eval(args[1]);

    tmp1 = (args[0])->hd;  tmp2 = (args[1])->hd;

    x->tag = tlist;
    x->hd = tmp1 + reduce_snd (tmp2);

    y = x->tl = args[0];
    y->tag = tapp;
    y->fun = reduce;
    y->arg = (struct rnode **) args[1];

    /* the tail remains the same */
    ((struct rnode *) (y->arg))->hd = reduce_fst (tmp2);
}
```

```
FILE:  edigits.c
----------------

/*  A C rewrite of David Turner's e digits Miranda program:
edigits = 2 : convert (repeat 1)
convert x = (hd x'):convert (tl x')
            where x' = norm 2 (0:map (10*) x)
norm c (d:e:x) = d + e div c: modc (c+1) (e:x),         if e mod c + 9 < c
               = d + e' div c: e' mod c : x',           otherwise
                 where
                 (e':x') = norm (c+1) (e:x)
modc c (a:x) = a' mod (c-1) : x'
               where
               (a':x') = norm c (a:x)
*/

#include <stdio.h>
#include "reals.h"

void norm (struct rnode *a, struct rnode **b);

void eval (struct rnode *x)
{

/* if x is a list, we leave it alone; otherwise, we evaluate one term */

    if (x->tag == tapp)
       (*x->fun) (x, x->arg);
}
```

```
void print (struct rnode *x)
{
    setbuf(stdout, NULL);

    for (; x->tl != NULL ; x = x->tl)
    {
       eval(x);
       printf("%d", x->hd);
    }
}

void mult10 (struct rnode *x, struct rnode *farg)
{
    struct rnode *tmp;

    eval(farg);

    x->tag = tlist;
    x->hd = (farg->hd)*10;

    tmp = x->tl = new_rnode();
    tmp->tag = tapp;
    tmp->fun = mult10;
    tmp->arg = farg->tl;

}

// modc c (a:x) = a' mod (c-1) : x'
//                where
//                (a':x') = norm c (a:x)
```

```
void modc (struct rnode *x, struct rnode *farg[])
{
   int c;
   struct rnode *args[2];
   struct rnode *tmp;

   c = (int) farg[0];

   args[0] = (struct rnode *) c;
   args[1] = farg[1];

   tmp = new_rnode();
   tmp->tag = tapp;
   tmp->fun = norm;
   tmp->arg = &args;

   eval(tmp);

   x->tag = tlist;
   x->hd = mod ((tmp->hd),c-1);
   x->tl = tmp->tl;

   free(tmp);
}

void norm (struct rnode *x, struct rnode **farg)
{
    struct rnode *next;
    struct rnode **args;
```

```
int c, d, e, newe;

c = (int) farg[0];

eval(farg[1]);
d = (farg[1])->hd;

eval((farg[1])->tl);
e = ((farg[1])->tl)->hd;

args = (struct rnode **) malloc (2 * sizeof (struct rnode *));
args[0] = (struct rnode *) (c+1);
args[1] = (farg[1])->tl;

next = new_rnode();
next->tag = tapp;
next->fun = modc;
next->arg = args;

x->tag = tlist;

if (mod(e,c)+9 < c)
{
  x->hd = d + div(e,c);
  x->tl = next;
}
else
{
  next->fun = norm;
  eval (next);
```

```
      newe = next->hd;
      x->hd = d + div(newe,c);
      x->tl = new_rnode();
      (x->tl)->tag = tlist;
      (x->tl)->hd = mod(newe,c);
      (x->tl)->tl = next->tl;
    };
}

void convert (struct rnode *x, struct rnode *farg)
{
    struct rnode *newx, *y, *tmp;
    struct rnode **norm_args;
    struct rnode **newx_tail;

// y = 0 : map (10*) x
    y = new_rnode();
    y->tag = tlist;
    y->hd = 0;
    y->tl = new_rnode();
    (y->tl)->tag=tapp;
    (y->tl)->fun=mult10;
    (y->tl)->arg=farg;

    norm_args = (struct rnode **) malloc (sizeof(int)+sizeof(struct rnode *));
    norm_args[0] = (struct rnode *) 2;
    norm_args[1] = y;

// newx = x'
    newx = new_rnode();
```

```
    (newx)->tag = tapp;
    (newx)->fun = norm;
    (newx)->arg = norm_args;

    eval(newx);

    x->tag = tlist;
    x->hd = newx->hd;
    x->tl = new_rnode();

    (x->tl)->tag = tapp;
    (x->tl)->fun = convert;
    (x->tl)->arg = newx->tl;
}

struct rnode *ones, *edigits;

int main ()
{
    ones = new_rnode();
    ones->tag = tlist;
    ones->hd = 1;
    ones->tl = ones;

    edigits = new_rnode();
    edigits->tag = tapp;
    edigits->fun = convert;

    edigits->arg = ones;
```

```
    printf("2.");
    print(edigits);

    return 0;
}
```

```
OUTPUT OF THE EDIGITS.C PROGRAM
-------------------------------

$ ./edigits
2.7182818284590452353602874713526624977572470936999595749669676277240766303
3535475945713821785251664274274663919320030599218174135966290435729003342
95260595630738132328627943490763233829880753195251019011573834187930702154
0891499348841675092447614606680822648001684774118537423454424371075390777
44992069551702761838606261331384583000752044933826560297606737113200709328
70912744374704723069697720931014169283681902551510865746377211125238978442
505695369677078544996996794686445490598793163688923009879312773617821542
4999229576351482208269895193668033182528869398496465105820939239829488793
32036250944311730123819706841614039701983767932068328237646480429531180232
8782509819455815301756717361332069811250996181881593041690351598888519345
807273866738589422879228499892086805825749279610484198444363463244968487
56023362482704197862320900216099023530436994184914631409343173814364054625
315209618369088870701676839642437814059271456354906130310720851038375051
01157477041718986106873969655212671546889570350354021234078498193343210681
701210056278802351930332247450158539047304199577770935036604169973297250
88687696640355570716226844716256079882651787134195124665201030592123667719
43252786753985589448969709640975459185695638023637016211204774272283648961
34225164450781824423529486363721417402388934412479635743702637552944483379
98016125492278509257782562092622648326277933386566481627725164019105900491
64499828931505660472580277863186415519565324425869829469593080191529872117
25563475463964479101459040905862984967912874068705048958586717479854667757
57320568128845920541334053922000113786300945560688166740016984205580403363
79537645203040243225661352783695117788386387443966253224985065499588623428
18997077332761717839280349465014345588970719425863987727547109629537415211
15136835062752602326484728703920764310059584116612054529703023647254929666
```

Bibliography

[1] Aberth, O., *Computable Analysis*, McGraw-Hill, New York, 1980

[2] Agrawal, D. P., "Arithmetic Algorithms in a Negative Base", *IEEE Trans. Comput.*, Vol. C-24, No. 10, Oct. 1975, pp. 998-1000

[3] Alefeld, G. and Herzberger, J., *Introduction to Interval Computations*, Academic Press, New York, 1983

[4] Avizienis, A., "Binary-compatible signed-digit arithmetic", *Proc. AFIPS Fall Joint Comp. Conf.*, 1964, pp. 663-672

[5] Avizienis, A., "Signed-digit number representations for fast parallel arithmetic", *IRE Trans. El. Comp.*, Vol. EC-10, No. 3, Sept. 1961, pp. 389-400

[6] Bailey, D. H., "Algorithm 719: Multiprecision Translation and Execution of FORTRAN Programs", *ACM Trans. Math. Software*, 19(3), 1993, pp. 288-319

[7] Bailey, David, Borwein, Peter and Plouffe, Simon, "On the rapid computation of various polylogarithmic constants", *Math. Comp.*, 66 (1997), pp. 903-913

[8] Beckmann, P., *A History of Pi*, St. Martin's Press, New York, 1971

[9] Bishop, Errett and Bridges, Douglas, "Constructive Analysis", Springer-Verlag, Berlin, 1985

[10] Blum, L., Shub, M. and Smale, S., "On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines", *Bull. Amer. Math. Soc.*, Vol. 21, No. 1, July 1989, pp. 1-46

[11] Boehm, H.-J., Cartwright R., et al., "Exact Real Arithmetic: A Case Study in Higher Order Programming", *Proceedings 1986 ACM Confrence on LISP and Functional Programming*, ACM Press (August 1986), pp. 162-163

[12] Boehm, Hans and Cartwright, Robert, "Exact Real Arithmetic: Formulating Real Numbers as Functions", in *Research Topics in Functional Programming*, (ed) D. A. Turner, Addison-Wesley, 1990

[13] Borwein, J. M., Borwein P. B. and Bailey, D.H., "Ramanujan, Modular Equations, and Approximations to Pi, or How to Compute One Billion Digits of Pi", *American Mathematical Monthly*, March 1989, pp. 201-219. Also available from the URL: `http://www.cecm.sfu.ca/personal/pborwein/`.

[14] Brent, R. P., "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, vol. 23 (1976), pp. 242-251

[15] Briggs, K., "A precise calculation of the Feigenbaum constants", *Math. Comput.*, Vol. 57, 1991, pp. 435-439

[16] Brouwer, L.E.J., "De Onbetrouwbaarheid der Logische Principes" (The untrustworthiness of the principles of logic), *Tijdschrift voor Wijsbegeerte*, Amsterdam, Vol. 2, 1908, pp. 152-158

[17] Cantor, Georg, *Contributions to the Founding of the Theory of Transfinite Numbers* (1895, 1897), Translated by P Jourdain, Open Court, Chicago, 1915

[18] Cantor, Georg, *Gesammelte Abhandlungen*, Springer, Berlin, 1932

[19] Church, Alonzo, "An Unsolvable Problem of Elementary Number Theory", *American J. of Math.*, Vol. 58, 1936, pp. 345-363

[20] Dedekind, Richard, *Essays on the Theory of Numbers*, Authorised translation by Prof. W W Beman, The Open Court Publishing Company, La Salle, Illinois, 1948

[21] Dedekind, Richard, *Stetigkeit und Irrationale Zahlen*, Vieweg, Braunschweig, 1872

[22] Denning, P.J., Dennis J.B. and Qualitz J.E., *Machines, Languages, and Computation*, Prentice-Hall, Inc., 1978

[23] de Regt, M. P., "Negative Radix Arithmetic", *Comp. Design*, Vol. 16, May 1967, pp. 52-63

[24] di Gianantonio, P., *A functional approach to real number computation*, PhD thesis, University of Pisa, 1993

[25] di Gianantonio, P., "Real number computability and domain theory", *Information and Computation*, 127 (1), May 1996, pp. 11-25

[26] Edalat, A. and Potts, P. J., "A New Representation for Exact Real Numbers", Electronic Notes in Theoretical Computer Science, 6 (1997), `URL: http://www.elsevier.nl/locate/entcs/volume6.html`

[27] Ercegovac M. D., "On-line Arithmetic: an Overview", SPIE Vol. 495, *Real Time Signal Processing VII*, 1984, pp. 86-93

[28] Escardó, M. H., "PCF extended with real numbers", *Theoretical Computer Science*, 162 (1), May 1996, pp. 79-115

[29] Eve, J., "The evaluation of polynomials", *Num. Math*, vol. 6 (1964), pp. 17-21

[30] Feigenbaum, M. J., "Quantitative universality for a class of nonlinear transformations", *J. Stat. Phys.*, Vol. 19, 1978, pp. 25-52

[31] Fike, C.T., "Methods of evaluating polynomials in function evaluation routines", *Comm. ACM*, vol. 10 (1967), pp. 175-178

[32] Friedman, H., *Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory*, Logic Colloquium 1969, (R. O. Gandy and Yates, eds.) C.M.E., North-Holland, Amsterdam, 1971, pp. 361-390

[33] Friedman, H. and Ko, K., "Computational complexity of real functions", *J. Theoret. Comp. Sci.*, Vol. 20, 1982, pp. 323-352

[34] Gosper, W., "Continued Fraction Arithmetic", HAKMEM Item 101B, MIT Artificial Intelligence Memo 239, MIT (1972)

[35] Heath, T. L., trans., "The Works of Archimedes", in Robert M. Hutchins, ed., *Great Books of the Western World*, vol. 11, Encyclopaedia Britannica, 1952, pp. 447–451.

[36] Heyting, A., *Intuitionism, an Introduction, Studies in Logic and Foundations of Mathematics*, North-Holland, Amsterdam, 1966

[37] Holmes, W. N., "Representation for Complex Numbers", *IBM Jl Res. Develop.*, Vol. 22, No. 4, July 1978, pp. 429-430

[38] Jones, R. and Lins, R., *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley and Sons, 1996

[39] Kaganovsky, A., "Computing with Exact Real Numbers in a Radix-r System", Electronic Notes in Theoretical Computer Science, Vol. 13 (1998)

[40] Kaganovsky, A., "Exact Complex Arithmetic in an Imaginary Radix System", UKC Technical Report 9-99, Computing Laboratory, University of Kent at Canterbury, 1999

[41] Kahan, W., "Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit", Chapter 7 in *The State of the Art in Numerical Analysis*, ed. by M. Powell and A. Iserles, Oxford (1987), pp. 165-211

[42] Kleene, S.C., *Introduction to Metamathematics*, New York, Amsterdam, and Groningen, 1952

[43] Knuth, D. E., "An Imaginary Number System", *Commun. ACM*, Vol. 3, No. 4, Apr. 1960, pp. 245-247

[44] Knuth, Donald E., "Mathematics and Computer Science: Coping with Finiteness", *Science*, Vol. 194 (17 December 1976), No. 4271, pp. 1235-1242.

[45] Knuth, D., *The Art of Computer Programming*, Vol.2 (Seminumerical Algorithms), Second Edition, Addison-Wesley, Reading, MA, USA, 1981

[46] Kornerup, P. and Matula, D.W., "Finite precision lexicographic continued fraction number systems", In *Proc. 7th IEEE Symposium on Computer Arithmetic*, IEEE Computer Society Press (1985), pp. 207-214

[47] Lang, Serge, *Algebra*, (3rd Ed.), Addison-Wesley Pub. Co., 1992

[48] Lang, Serge, *Complex Analysis*, 4th Ed., Graduate Texts in Mathematics (103), Springer-Verlag, 1999

[49] Lyusternik, L. A., Chervonenkis, O. A., and Yanpol'skii, A. R., *Handbook for Computing Elementary Functions*, Pergamon Press, New York, 1965 (English translation by G. J. Tee)

[50] Mazenc, Christophe, "On the Redundancy of Real Number Representation Systems", Ecole Normale Supérieure de Lyon, Research Report No. 93-16, May 1993

[51] Ménissier-Morain, V., "Arbitrary Precision Real Arithmetic: Design and Algorithms", Unpublished manuscript

[52] Moore, R. E., *Methods and Applications of Interval Analysis*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1979

[53] Muller, J.-M. "Arithmétique des ordinateurs", *Etudes et recherches en informatique*, Masson, 1989

[54] Myhill, John R., "A Complete Theory of Natural, Rational, and Real Numbers", *The Journal of Symbolic Logic*, Vol. 15, No. 3, Sept. 1950, pp. 185-196

[55] Myhill, John, "What is a real number?", *American Mathematical Monthly*, September 1972, pp. 748-754

[56] *Numerical Recipes in C: The Art of Scientific Computing* (Second Edition), by W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, Cambridge University Press, 1995

[57] Omondi, Amos R., textitComputer Arithmetic Systems: Algorithms, Architecture and Implementations, Prentice Hall, 1994

[58] O'Reilly, T. J., "A Positional Notation for Complex Numbers", *IEEE Computer Society Repository, R74-169*, 12 pp. (1974), IEEE Computer Society Publications Office, Long Beach, CA 90803

[59] Pawlak, Z., "An Electronic Digital Computer Based on the '—2' System", *Bull. de l'Acad. Polonaise des Sciences, Série des Sci. Tech.*, Vol. 7, No. 12, 1959, pp. 713-721

[60] Pawlak, Z. and Wakulicz, A., "Use of Expansions with a Negative Basis in the Arithmometer of a Digital Computer", *Bull. Acad. Polon. Sci., Classe III*, Vol. 5, March 1957, pp. 233-236

[61] Penney, W., "A 'Binary' System for Complex Numbers", *Jl ACM*, Vol. 12, No. 2, Apr. 1965, pp. 247-248

[62] Pixley, C. P., "Demand-Driven Arithmetic", Burroughs Corporation Austin Research Center, Internal Report ARC 82-18, Nov. 1982

[63] Plume, D., "A Calculator for Exact Real Number Computation", University of Edinburgh, Dept of Computer Science and Artificial Intelligence, 4th Year Project Report, 1998. URL: `ftp://ftp.tardis.ed.ac.uk/users/dbp/report.ps.gz`

[64] Pour-El, Marian B., and Richards, Ian J., *Computability in Analysis and Physics*, Berlin, London: Springer-Verlag, 1989

[65] Pour-El, M.B. and Richards, I., "Computability and noncomputability in classical analysis", *Trans. Amer. Math. Soc.*, Vol. 275, No. 2, Feb. 1983, pp. 539-560

[66] Rice, H.G., "Classes of recursively enumerable sets of positive integers and their decision problems", Trans. Amer. Math. Soc., Vol. 74 (1953), No. 2, pp. 358-366

[67] Rice, H.G., "Recursive real numbers", *Proc. Amer. Math. Soc.*, Vol. 5 (October 1954), No. 5, pp. 784-791.

[68] Robertson, J. E., "A New Class of Digital Division Methods", *IRE Trans. El. Comp.*, Vol. EC-7, Sept. 1958, pp. 218-222

[69] Robinson, R.M., Review, *J. Symbolic Logic*, Vol. 16, 1951, p. 282

[70] Salamin, E., "Computation of Pi using Arithmetic-Geometric Mean", *Mathematics of Computation*, vol. 30 (1976), pp. 565–570

[71] Šanin, N.A., *Constructive Real Numbers and Constructive Function Spaces*, Transl. Math. Monographs, Vol. 21, Amer. Math. Soc., Providence, R.I., 1968

[72] Sankar, P. V., Chakrabarti, S., and Krishnamurthy, E.V., "Arithmetic Algorithms in a Negative Base", *IEEE Trans. Comput.*, Vol. C-22, No. 2, Feb. 1973, pp. 120-125

[73] Schwarz, J., "Implementing Infinite Precision Arithmetic", *Proc. 9th Symposium on Computer Arithmetic*, September 6-8, 1989, pp. 10-17

[74] Scott, Norman R., *Computer Number Systems and Arithmetic*, Prentice-Hall, Inc., 1985

[75] Simpson, A., "Lazy Functional Algorithms for Exact Real Functionals", in *Mathematical Foundations of Computer Science 1998*, Springer LNCS 1450, pp. 456-464, 1998

[76] Smith, D. M., "Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic", *ACM Trans. Math. Software*, 17(2), 1991, pp. 272-283

[77] Smith, D. M., "A Multiple-Precision Division Algorithm", *Mathematics of Computation*, Vol. 65, No. 213, Jan. 1996, pp. 157-163

[78] Songster, G. F., "Negative-Base Number-Representation Systems", *IEEE Trans. Electr. Comp.*, June 1963, pp. 274-277

[79] Specker, E., "Nicht Konstructiv beweisbare Sätze der Analysis", *J. Symbolic Logic*, Vol. 14, 1949, pp. 145-158

[80] Stromberg, Karl R., *Introduction to Classical Real Analysis*, Wadsworth, Inc., 1981

[81] *The Universal Turing Machine: A Half-Century Survey*, edited by Rolf Herken, Oxford University Press, 1988

[82] Trivedi, K.S. and Ercegovac, M.D., "On-line algorithms for division and multiplication", *IEEE Trans. Comp.*, Vol. C-26, No. 7, 1977, pp. 681-687

[83] Turing, Alan M., "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proc. London Math. Soc.*, Ser. 2, Vol. 42, Nov. 12, 1936, pp. 230-265

[84] Turing, Alan M., "On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction", *Proc. London Math. Soc.*, Ser. 2, Vol. 43, No. 2198, 1937, pp. 544-546

[85] Turner, D.A., "Miranda: A non-strict functional language with polymorphic types", Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France, September 1985 (Springer Lecture Notes in Computer Science, Vol. 201)

[86] Varga, R. S., *Scientific Computation on Mathematical Problems and Conjectures*, SIAM, Philadelphia, 1990

[87] Vuillemin, Jean E., "Exact Real Computer Arithmetic with Continued Fractions", *IEEE Transactions on Computers*, Vol. 39, No. 8, August 1990, pp. 1087-1105

[88] Wadel, L. B., "Negative Base Number Systems", *IRE Trans. Electr. Comp.*, Vol. EC-6, June 1957, p. 123

[89] Wiedmer, E., "Computing with Infinite Objects", *Theoretical Computer Science*, Vol. 10 (1980), pp. 133-155

[90] Wiedmer, E., "Exaktes Rechnen mit reellen Zahlen und anderen unendlichen Objekten", Diss. ETH 5975, Zurich (1977)

[91] Yuen, C. K., "On the Floating Point Representation of Complex Numbers", *IEEE Trans. Comput.*, Vol. C-24, No. 6, Aug. 1975, pp. 846-848

[92] Zaslavskii, I.D., *Some Properties of Constructive Real Numbers and Constructive Functions*, Amer. Math. Soc. Transl. (2), Vol. 57, 1966, pp. 1-84