# NOTE TO USERS

# ANALYTIC COMPUTATION OF THE PRIME-COUNTING FUNCTION

BY

WILLIAM FLOYD GALWAY

B.A., University of Utah, 1976
M.S., University of Utah, 1984

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

# UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

# GRADUATE COLLEGE

OCTOBER 2001

**date**

**WE HEREBY RECOMMEND THAT THE THESIS BY**

WILLIAM FLOYD GALWAY

**ENTITLED** ANALYTIC COMPUTATION OF THE PRIME-COUNTING FUNCTION

**BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF** DOCTOR OF PHILOSOPHY

Harold G Diamond

**Director of Thesis Research**

**Head of Department**

**Committee on Final Examination†**

**Chairperson**

**†Required for doctor's degree but not for master's.**

**O-517**

# Abstract

The main topic of this dissertation is the Lagarias-Odlyzko analytic algorithm for computing $\pi(x)$—the number of primes less than or equal to $x$. This algorithm is asymptotically the fastest known algorithm for the computation of $\pi(x)$. It uses numerical integration of a function related to the Riemann zeta function, $\zeta(s) := \sum_{n=1}^{\infty} n^{-s}$, in combination with a summation involving a "kernel" function evaluated at prime powers near $x$.

Our work resolves many issues left untreated in the original paper by Lagarias and Odlyzko and makes several original contributions—some of which have applications in other areas. In this dissertation we

- introduce a kernel function which appears to be more effective than one suggested by Lagarias and Odlyzko;
- perform a careful analysis of various sources of truncation error;
- give choices of parameters which bound the truncation error while keeping computation to a minimum;
- develop two new methods for enumerating primes in intervals, which require much less memory than previously known sieving methods and which are much faster than methods which test primality of single numbers;
- describe a new method for computing $\zeta(s)$ which gives more accurate values with less complexity than classical methods.

In Honor of my Father
Floyd I. Galway
and to the memory of my Mother
Desma H. Galway

# Acknowledgments

I'm grateful for the advice, support, encouragement, and friendship of the many individuals—and of a few organizations—who contributed to this dissertation. In many cases I give further details on their contributions in the acknowledgments at the ends of individual chapters. I have also tried to place entries for each contributor in the **Index and Glossary** which begins on page 163.

First, I thank my advisor: Harold G. Diamond, who has been an unflagging guide. His support and his excellent mathematical judgment–especially his judgment of when a result is good-enough for the task at hand–have helped me to express myself mathematically. I also thank Harold and his wife Nancy for graciously hosting me, twice, as a houseguest during the final stages of completing this dissertation.

I also thank the other members of my dissertation committee: Bruce Berndt, Adolf Hildebrand, Andrew Odlyzko, and Frank Stenger.

Professor Berndt and Professor Hildebrand were always happy to discuss issues as they arose during my research. Some of their specific contributions are detailed in the Acknowledgments for Chapters 5 and 6. Professor Hildebrand also advised me on some of the finer points of working with LaTeX.

Both before and after joining my committee, Andrew Odlyzko always found time in his busy schedule to advise me on fine points of the algorithms which formed the starting point for this dissertation: the Lagarias-Odlyzko analytic algorithm for computing $\pi(x)$; the Meissel, Lehmer, Lagarias, Miller, Odlyzko (extended Meissel-Lehmer) algorithm for computing $\pi(x)$; and the Odlyzko-Schönhage algorithm for computing $\zeta(s)$.

Long before he joined my committee, Professor Stenger was always happy to discuss the questions I posed to him on numerical analysis. Although the analyses in this dissertation are self contained, and I rarely cite Professor Stenger's work, I believe that the approach to numerical quadrature used here fits under the rubric of "sinc methods"—a field which Professor Stenger is largely responsible for developing.

Both Jeff Lagarias and Victor Miller gave advice, historical background on algorithms, and encouragement for this research.

Dan Bernstein introduced me to some of the key ideas leading to my development of the dissected sieve, as described in Chapter 5. Carl Pomerance provided the inspiration to do the research leading to the hybrid sieve, described in Chapter 6.

Richard Crandall gave several suggestions, which I have used, on the computation of special functions. He also asked probing questions about preliminary versions of this work, and I hope that my exposition has been improved in responding to those questions.

Some of the research described here was completed while I was supported as a fellow at the Pacific Institute for Mathematical Sciences (PIMS) at Simon Fraser University in Burnaby, British Columbia, Canada.

During the research for and the writing of this dissertation I made heavy use of computers. I thank the many system administrators and their staff who have supported the smooth operation of those resources: Robert Zeh, Mike Hollyman, Carl Freeland, Jason Hoos, Ken Hamer, David Fiske, Bob Berry, Mary Stevens, Jennifer Shannon, Jonathan Manton, and Patrick Szuta at the University of Illinois; and Brent Kearney at the Pacific Institute for Mathematical Sciences.

Sabrina Hwu and Teresa Johnson, at the Center for Reliable and High Performance Computing, University of Illinois Urbana-Champaign, provided information on contemporary computer architectures and cache memories.

Shortly after the death of my mother, while I was preparing to re-enter the academic world, my friend John Anderson encouraged and assisted me in my application for entrance to the University of Illinois.

My wife, Miriam Paschetto, has supported me both emotionally and financially. She also did the bulk of the packing to move our household first to the Pacific Coast, and later to the Atlantic Coast. I thank her for her time, work, and support, and for both her patience and her impatience as I neared the completion of this work.

Most of the calculations reported here were performed using programs written in the C programming language. I also used the *Mathematica*® software system[1] and the PARI/GP package developed by Henri Cohen, Karim Belebas, and many other collaborators. PARI/GP is currently available at `http://pari.math.u-bordeaux.fr/`.

[1] *Mathematica* is a registered trademark of Wolfram Research, Inc.

The layout of this dissertation is defined by a LaTeX document class which I developed from David Hull's `uiucthesis.sty`. Algorithms are presented using Carsten Heinz's **listings** package for LaTeX.

Many of the figures in this dissertation were created using *Mathematica*. In many cases the figures were further annotated using the Unix utility `xfig` and/or LaTeX's `PSfrag` package.

Some of the computations described in Chapter 6 made use of the Condor system[2] for "high throughput computing" on a distributed system of computers.

Finally, I thank the following friends, colleagues, and correspondents: Hiro Asari, David Bailey, Rob Ballantyne, Michael Bennett, Robert Bennion, Maarten Bergvelt, Jonathan Borwein, Peter Borwein, David Bradley, Richard Brent, Jared Bronski, Jeff and Sharon Brueggeman, Imin Chen, Stephen Choi, Darrin Doud, Helaman Ferguson, Gilbert Gosseyn, Gergely Harcos, Steve Harding, Martin Huxley, François Morain, Richard Pinch, Ekkehart Vetter, Ulrike Vorhauer, Ron Weeden, and Alexandru Zaharescu.



[2]The Condor Software Program (Condor) was developed by the Condor Team at the Computer Sciences Department of the University of Wisconsin-Madison. All rights, title, and interest in Condor are owned by the Condor Team.

# Contents

# List of Tables

# List of Figures

# 1 Introduction

## 1.1 Background

The function $\pi(x)$, (Figure 1.1) counts the number of primes up to $x$. More specifically,

$$\pi(x) := \sum_{p \leq x} 1,$$

where the sum is taken over primes.

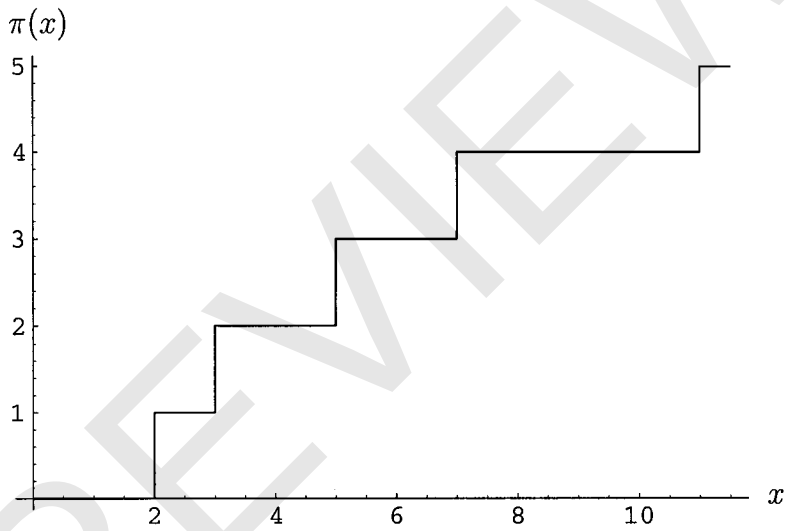


Figure 1.1: The function $\pi(x)$, $0 \leq x \leq 11.5$

In 1982 [LO84] Jeff Lagarias and Andrew Odlyzko described two new algorithms for computing $\pi(x)$. One method, their "analytic" algorithm, uses numerical integration of a function related to the Riemann zeta function, $\zeta(s) := \sum_{n=1}^{\infty} n^{-s}$, in combination with summation of a function evaluated at prime powers "near" $x$. ("Prime powers" means all numbers of the form $p^m$, $p$ prime, $m \geq 1$.) In a later paper, after Odlyzko and A. Schönhage [OS88] had developed a fast method for computing $\zeta(s)$, Lagarias and Odlyzko showed that their analytic algorithm could compute $\pi(x)$ in $x^{1/2+\epsilon}$ time and $x^{1/4+\epsilon}$ space [LO87]. (We give more precise definitions of "time" and "space" in Section 1.5.)

The other method of Lagarias and Odlyzko—the extended Meissel-Lehmer method—with later improvements by Marc Deléglise and Joel Rivat, computes $\pi(x)$ in $O(x^{2/3}/\ln^2 x)$ time and $O(x^{1/3}\ln^3(x)\ln\ln(x))$ space [DR96a]. Lagarias, Miller, and Odlyzko used the original version of this algorithm to compute values of $\pi(x)$ up to $\pi(4\cdot 10^{16})$ [LMO85]. Deléglise and Rivat improved the algorithm and continued computations up to $\pi(10^{18})$, and more recently to $\pi(10^{20})$ [DR96a, DR96b].

The extended Meissel-Lehmer algorithm has been further improved by Xavier Gourdon [Gou01a]. His most important improvement has been to modify the algorithm so that it is well suited to distributed computation, with little communication needed between processors. Gourdon has implemented an ongoing project of computing $\pi(x)$ using resources provided by several contributors on the Internet. As of March 2001, computation had proceeded through $\pi(4\cdot 10^{22})$ [Gou01b]. While attempting to compute $\pi(10^{23})$, Gourdon found an error in his program—two slightly different computations gave two different values for $\pi(10^{23})$. As of early 2004, $\pi(4\cdot 10^{22})$ appears to be the "record computation" of $\pi(x)$.

Until the research of this dissertation, little work has gone towards carefully analyzing the analytic algorithm—other than the few papers by Lagarias and Odlyzko cited above, and an analysis given by Ekkehart Vetter in his Diplomarbeit [Vet91, Chapter 2]. Although the analytic method has better asymptotic running time than other methods, the $O$-constant is expected to be large.

The following crude argument suggests that it is only for very large values of $x$ that the analytic algorithm is faster than the extended Meissel-Lehmer algorithm. Ignoring factors of $x^\epsilon$ we assume that the running time for the extended Meissel-Lehmer algorithm has the form $a_1 x^{b_1}$, and that the running time for the analytic algorithm has the form $a_2 x^{b_2}$. This implies that the latter will be faster for $x \geq (a_2/a_1)^{1/(b_1-b_2)}$. If we assume that $b_1 = 2/3$ and that $b_2 = 1/2$, this implies a crossover point at $x = (a_2/a_1)^6$. However, fitting the timing data given in [DR96a] gives $b_1 \approx 0.625$ (Figure 1.2). If we further assume (arbitrarily, but perhaps realistically) that $b_2 \approx 0.525$ near the crossover point, we find crossover near $x = (a_2/a_1)^{10}$.

Without an implementation it is difficult to estimate what $a_2/a_1$ might be, but values from 100 to 10,000 seem plausible, implying crossover in the range $10^{12} \leq x \leq 10^{40}$. In any case, it is clear that a careful implementation is

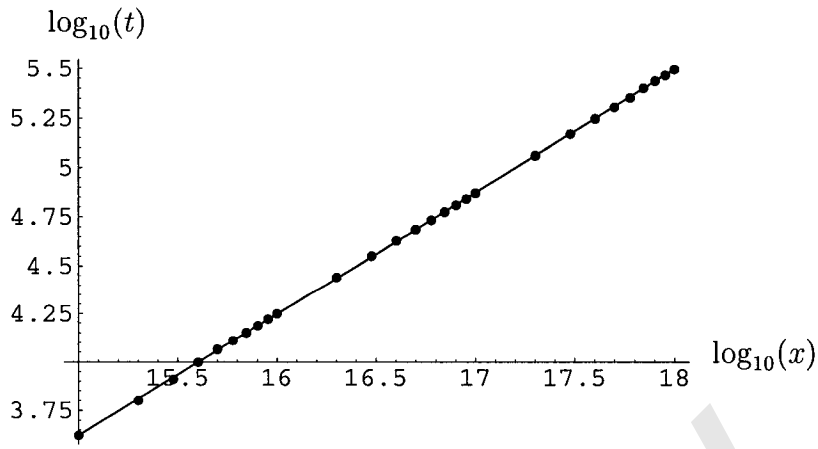


Figure 1.2: Log-Log fit to Deléglise-Rivat timings
$\log_{10}(t) = -5.768 + 0.626 \log_{10}(x)$
($t$ in seconds)

required to bring the crossover point into a practical range. This dissertation attempts to give nearly optimal solutions for the tasks that must be solved to produce an implementation.

## 1.2 Notational conventions

Many of our notational conventions are outlined in Table 1.1 on the following page. Although most of our notation is standard, there are a few fine points and nonstandard conventions which we describe below.

Given $f(z)$ and $g(z)$, possibly complex valued, we write $f(z) = O(g(z))$ (equivalently, $f(z) \ll g(z)$, $g(z) \gg f(z)$) if there is a constant $C > 0$ (an *O-constant*) such that $|f(z)| \leq C\,|g(z)|$ over some domain (that should be clear from the context). We write $f(z) \asymp g(z)$ to denote that $f(z)$ and $g(z)$ are of the same order, i.e., $f(z) \ll g(z)$ and $f(z) \gg g(z)$. A subscript in the notation indicates a variable that the constant depends upon. For example, $f(z) = O_\rho(g(z))$ means that there is a function $C(\rho)$, independent of $z$, for which $|f(z)| \leq C(\rho)\,|g(z)|$.

We use the nonstandard notation $f(z) = C\,O(g(z))$ to specify an explicit $O$-constant. Our meaning is that $|f(z)| \leq C\,|g(z)|$ for the given $C$. We write $f(z) = h(z) + O(g(z))$ to mean $f(z) - h(z) = O(g(z))$, and similarly

| Notation | Meaning |
|---|---|
| ■ | Terminates a proof, or an argument supporting a conjecture. |
| □ | Terminates a definition or remark. |
| $:=$ | Equality defining the left side. |
| $=:$ | Equality defining the right side. |
| $f(x)\|_a^b$ | Denotes $f(b) - f(a)$. Also written $f(x)\|_{x=a}^{x=b}$. |
| $\lfloor x \rfloor$ | Greatest integer $\leq x$. |
| $\lceil x \rceil$ | Least integer $\geq x$. |
| $x \bmod b$ | The value $x - b\lfloor x/b \rfloor$. The expression $x \bmod b = y$ often implies a computation yielding $y$ as its value. In contrast, $x \equiv y \pmod b$ states that a congruence holds, without implying computation. |
| $\|\mathcal{S}\|$ | Cardinality of a set or sequence $\mathcal{S}$. |
| $\overline{z}$ | Complex conjugate of $z \in \mathbb{C}$. |
| $\mathrm{Re}(z)$ | Real part of $z \in \mathbb{C}$. |
| $\mathrm{Im}(z)$ | Imaginary part of $z \in \mathbb{C}$. |
| $\ln(z)$ | Natural logarithm of $z$. |
| $\mathrm{Ln}(z)$ | Principal branch of $\ln(z)$, satisfies $-\pi < \mathrm{Im}\,\mathrm{Ln}(z) \leq \pi$, $e^{\mathrm{Ln}(z)} = z$. |
| $\log_b(x)$ | Logarithm of $x$ to the base $b$, $\log_b(x) = \ln(x)/\ln(b)$. |
| $\mathbb{N}$ | The set of strictly positive integers, $\mathbb{N} = \{1, 2, \dots\}$. |
| $\mathbb{Z}_0$ | The set of non-negative integers, $\mathbb{Z}_0 = \{0, 1, \dots\}$. |
| $\mathcal{P}$ | The set of prime numbers. |
| $p$, $\ell$ | Unless stated otherwise, $p \in \mathcal{P}$ and $\ell \in \mathcal{P}$. E.g., $\sum_p \dots$ denotes a sum over all primes. |
| $\epsilon$ | A positive number that can be taken arbitrarily close to zero. Not to be confused with $\varepsilon$. |
| $\varepsilon$ | Denotes a fixed error bound, $\varepsilon > 0$. We often write $z + \varepsilon\, O(1)$ to denote a floating point approximation to a number, as discussed in Section 1.4. |
| § | Indicates a section number in citations, as in "§10.11.II". |
| $\mathrm{sgn}(x)$ | The "sign" or "signum" function, $\mathrm{sgn}(x) := -1, 0, 1$ for $x < 0$, $x = 0$, $x > 0$, respectively. |
| $\sum_{n \leq x}{}' \dots$ | Sum with a weighting of $1/2$ if $n = x$. Similarly for $\sum_{p \leq x}{}' \dots$ and for $\sum_{p^m \leq x}{}' \dots$. |

Table 1.1: Notational Conventions

$f(z) = h(z) + C\,O(g(z))$ means $f(z) - h(z) = C\,O(g(z))$. In particular,

$$f(z) = h(z) + \varepsilon\,O(1) \tag{1.1}$$

means that $h(z)$ approximates $f(z)$ with an error bounded by $\varepsilon$.

Given $g(z)$, possibly complex valued, which satisfies $g(z) \neq 0$ in a neighborhood of $z_0$, we write $f(z) \sim g(z)$ as $z \to z_0$ to mean $\lim_{z\to z_0} f(z)/g(z) = 1$, and write $f(z) = o(g(z))$ as $z \to z_0$ to mean $\lim_{z\to z_0} f(z)/g(z) = 0$. We also write $f(z) = h(z) + o(g(z))$ if $f(z) - h(z) = o(g(z))$. In many cases the $z_0$ should be clear from the context, and will not be specified.

For some functions, say $f(x)$ to be specific, we will only give definitions valid when $x > 0$. In such cases we let $f(0) := \lim_{\delta\to 0+} f(\delta)$ whenever the limit is well defined.

Many of our functions, such as $\phi(u; x, \lambda)$, depend on a "variable" $u$ and other "parameters" $x$, $\lambda$, etc., which will typically be fixed while $u$ varies. In such cases we may write $\phi(u)$ for $\phi(u; x, \lambda)$, with the understanding that the parameters remain fixed.

The integrals used in this dissertation can be taken as Riemann-Stieltjes integrals. Although most of our sums and integrals converge absolutely, some are only conditionally convergent. To ensure that these are well defined, we use the conventions that

$$\sum_{n\in\mathbb{Z}} \cdots := \sum_{n=-\infty}^{\infty} \cdots := \lim_{N\to\infty} \sum_{n=-N}^{N} \cdots,$$

$$\int_{-\infty}^{\infty} \cdots dt := \lim_{T\to\infty} \int_{-T}^{T} \cdots dt.$$

Integrals of the form $\int_0^\infty \ldots du$ are best thought of as the improper integral $\int_{-\infty}^{\infty} \ldots d\tau$, under the change of variables $u := e^\tau$.

Empty sums and products (sums and products with no terms) always denote the additive or multiplicative identity, respectively. For example, $\sum_{k=1}^{0} k = 0$ while $\prod_{k=1}^{0} k = 1$.

The **Index and Glossary** beginning on page 163 serves as a guide to notation introduced in other sections of this dissertation.

## 1.3 Algorithmic notation

Much in the same way that we number theorems, definitions, etc., we identify our algorithms with a numeric label of the form $C.N$, where $C$ denotes the chapter where the algorithm is defined, and $N$ numbers the algorithm within the chapter. We also give each algorithm an alphanumeric name. Algorithm 1.1 (`ExampleErfc`), on page 10, illustrates this convention. All the algorithms in this dissertation are indexed under their name, and under the entry for "Algorithms", in the **Index and Glossary**.

Algorithms are presented in a mixture of mathematical notation and notation based on the C and C++ computer languages.

We emphasize important conditions or truths with **assert** statements: which assert that a predicate holds without implying any computation. We start short comments with "//" which run to the end of the line. Longer comments may be interspersed between lines of an algorithm, as in the comments following line 2 of Algorithm 1.1 (`ExampleErfc`).

Assignment is denoted by $\leftarrow$, and $x$++ is shorthand for $x \leftarrow x + 1$. We may use assignments as values—for example to assign a value to $n$ and then compare that value against $x_2$ we could use the expression $(n \leftarrow u_1^2+u_2^2) \leq x_2$.

We write && for the conditional conjunction of boolean expressions. That is, in evaluating `expr1 && expr2` the second subexpression is evaluated only if `expr1` is `TRUE`, ultimately yielding the logical *and* of the two subexpressions. For unconditional conjunction (logical *and*) we write `expr1` $\wedge$ `expr2`.

We use the standard **while** and **for** looping constructs of the C language and also use nonstandard instances of **for** as illustrated below:

**for** $(n \leftarrow 1; n \leq 10; n{+}{+})$ ... // *Standard* **for** *loop.*
**for** $(1 \leq n \leq 10)$ ... // *Shorthand for the line above.*
**for** $(p \in [2, 100] \cap \mathcal{P})$ ... // *Loop over primes* $p$, $2 \leq p \leq 100$.

Of course, in addition to these looping constructs, we also use traditional mathematical notation such as $\sum_{m=1}^{M} \ldots$ and $\prod_{m=1}^{M} \ldots$ to indicate sums and products taken over a range of integers.

Without complete consistency, we follow some rules-of-thumb concerning the scope of variables and other names used in our algorithms. Variables passed as arguments can be assumed local to an algorithm, as can most other variables used within the algorithm. Names of algorithms such as `ExampleErfc` are treated as global, as are standard functions such as $\sin(z)$

and well-known constants such as $\pi = 3.14159\ldots$.

We do not use a rigorously typed language—instead, we use a variety of methods to indicate the type of a variable when we consider its type to be important. When we do not indicate the type it is because we believe the type to be either obvious or immaterial. Methods for indicating type include comments at the start of an algorithm, the choice of typeface used for a variable, or a *declaration* such as $x \in \mathbb{Z}$.

Without going into details of their representation, we assume that integers may have any magnitude and are represented exactly. Given $z \in \mathbb{Z}$ we call $z$ an $n$-bit integer provided $|z| < 2^n$. It follows from this convention that an $n$-bit integer is also an $N$-bit integer for any $N \geq n$. We assume that $z \in \mathbb{Z}$ can be represented using $O(\ln(2 + |z|))$ bits, from which it follows that an $n$-bit integer can be represented using $O(1 + n)$ bits.

We assume that a rational number, say $x/y$, is represented by a pair $x \in \mathbb{Z}$, $y \in \mathbb{N}$, not necessarily coprime. (We rarely use rationals, and the requirement that $\gcd(x, y) = 1$ is unnecessary for our purposes.)

Unless they are obviously elements of $\mathbb{Z}$ or $\mathbb{Q}$, real numbers are assumed to be approximated by floating point numbers, following conventions which are detailed in the next section, and which are defined as follows:

**Definition 1.1** Given $z \in \mathbb{R}$, we say that $z$ is a *floating point* number provided $z = 2^E M$, $E \in \mathbb{Z}$, $M \in \mathbb{Z}$.

We say that the pair $E \in \mathbb{Z}$, $M \in \mathbb{Z}$ define an $n$-bit floating point representation of $z = 2^E M$ provided $|M| < 2^n$ and $|E| = O(n)$. We do *not* require that $|M| \geq 2^{n-1}$. The $O$-constant in the bound on $|E|$ will remain unspecified but it could be deduced by careful analysis of the algorithms presented in this dissertation. □

**Remarks** Instead of saying that $z$ has an $n$-bit floating point representation, for ease of writing we are often willing to confuse $z$ with its representation and will say that $z$ is an $n$-bit floating point number if it has such a representation.

Assuming that an $n$-bit floating point number $z = 2^E M$ is represented by the pair $E$, $M$, it follows that $z$ requires $O\big(\ln(2+|E|+|M|)\big) = O(1+n)$ bits for its representation. We do not require that floating point numbers fit within the precision of any particular hardware, but instead allow arbitrary precision, following conventions spelled out in Section 1.4. □

As would be expected, unless they are obviously elements of $\mathbb{R}$, complex numbers are assumed to be approximated by numbers of the form $x + iy$, with $x$ and $y$ represented as floating point numbers. We call $z = x + iy$ an $n$-bit (complex) number if both $x$ and $y$ are $n$-bit floating point numbers.

## 1.4 Floating point numbers and error analysis

Given arbitrary $z \in \mathbb{R}$ or $z \in \mathbb{C}$, we make heavy use of the notation $z + \varepsilon\, O(1)$ to denote a number which differs from $z$ by an amount bounded by $\varepsilon$ in absolute value.

The "error term" denoted by $\varepsilon\, O(1)$ serves two, slightly different, purposes. One use is as a bookkeeping tool to document the accumulation of errors during a computation. These errors may arise both from truncation error, such as the error due to approximating integrals and infinite sums as finite sums; and from roundoff error, which is the error due to approximating arbitrary numbers with finite-precision floating point numbers.

When describing computations executed within an algorithm—as opposed to discussing computations within an **assert** statement, or within a theorem (say)—the inclusion of $\varepsilon\, O(1)$ in a sum has the further meaning of indicating the rounding of the sum to a certain number of bits of precision. More specifically, given $z \in \mathbb{R}$ we assume that a computed result of the form $z + \varepsilon\, O(1)$ denotes $2^E M$, with $E := \lfloor \log_2(\varepsilon) \rfloor$ and with $M$ chosen so that $\left|2^E M - z\right| < 2^E$, and thus $\left|2^E M - z\right| < \varepsilon$.

In this setting, note that we have $\left|2^E M - z\right|$ strictly less than $\varepsilon$. It follows that after executing a statement like

$z_1 \leftarrow (z + \varepsilon\, O(1)) + \varepsilon\, O(1)$;

we can conclude that $z_1 = z + \varepsilon\, O(1)$, and not just the weaker conclusion that $z_1 = z + 2\varepsilon\, O(1)$. More generally, when we round an already rounded expression to another precision we have

$$\left(z + \varepsilon_1\, O(1)\right) + \varepsilon_2\, O(1) = z + \max(\varepsilon_1, \varepsilon_2)\, O(1). \tag{1.2}$$

In the case of complex numbers, given $z = x + iy$ we assume that a computed result of the form $z + \varepsilon\, O(1)$ denotes the rounded result

$$\left(x + \frac{\varepsilon}{2} O(1)\right) + i\left(y + \frac{\varepsilon}{2} O(1)\right),$$

where the real and imaginary parts are floating point numbers.

Although not strictly necessary, when $z + \varepsilon\, O(1)$ denotes a floating point number we often use $\varepsilon$ of the form $\varepsilon = 2^E$ both to emphasize that $z$ is approximated by a floating point number, and because $\varepsilon$ of this form are themselves easily represented as floating point numbers.

By the conventions of Definition 1.1 on page 7, and the subsequent discussion, it follows that $z + \varepsilon\, O(1)$ is an $O(\ln(2 + |z/\varepsilon|))$-bit number—both for $z \in \mathbb{R}$ and for $z \in \mathbb{C}$; and provided that $\varepsilon \ll \ln(2 + |z/\varepsilon|)$, with an $O$-constant for the bound which is related in an obvious way to the $O$-constant mentioned in Definition 1.1. Our complexity analyses of algorithms will depend on these estimates of the number of bits in the representations of numbers, as explained further in Section 1.5.

In our presentation of algorithms we assume that all arithmetic operations are computed exactly and then rounded as indicated. In particular, we assume that operations are exact when they yield integers, rationals, or boolean values. This assumption extends to calculations which involve results outside the realm of $\mathbb{Z}$ or $\mathbb{Q}$. For example, we assume that boolean expressions such as $x_1^{1/m} \le p \le x_2^{1/m}$ are evaluated unambiguously provided $x_1$, $x_2$, $m$ and $p$ are in $\mathbb{N}$. Similarly, given $z = 2^E M$ we assume that expressions such as $\lceil \log_2(z) \rceil$ are computed exactly.

Our convention that computations are rounded from exact results implies what happens when we sum several rounded quantities, namely that

$$\sum_k (z_k + \varepsilon_k O(1)) = \sum_k z_k + \left( \sum_k \varepsilon_k \right) O(1), \tag{1.3}$$

under the standard assumption that all $\varepsilon_k > 0$. Computations such as those illustrated in (1.2) or (1.3) need not occur within a single expression—they would yield the same result even if the computations were spread over several statements in an algorithm. Note that Equation (1.3) would certainly not hold if our algorithms used floating-point arithmetic as implemented in contemporary hardware. We can assume that it holds since we assume that floating-point arithmetic, as defined in this section, adjusts the number of bits of precision as needed to ensure that our assumptions are met.

Note that our convention that computations are performed exactly and then rounded allows considerable leeway in the amount of detail we give